VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Synchronizace mobilního telefonu se službou Google Calendar

Mobile synchronization with Google Calendar Service

2012 Marek Friedrich

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Marek Friedrich**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Synchronizace mobilního telefonu se službou Google Calendar
Mobile Synchronization with Google Calendar Service

Zásady pro vypracování:

Cílem práce bude vytvořit aplikaci, která umožní synchronizovat mobilní telefon se službou Google Calendar. Aplikace by měla podporovat jak časové události, tak i úkoly. Dále by měla být schopna pracovat s více kalendáři v rámci jednoho uživatelského účtu s možností volby, které kalendáře chceme synchronizovat.

Během řešení postupujte podle následujících bodů:
1. Seznamte se a popište již existující řešení pro platformy Windows Mobile 6.5 a Windows Phone 7, které používají službu Google Calendar.
2. Vytvořte analýzu a návrh budoucí aplikace, která umožní obousměrnou synchronizaci položek mezi mobilním telefonem a službou Google Calendar.
3. Implementujte řešení pro platformu Windows Mobile 6.5, tak aby případná migrace na Windows Phone 7 byla co nejjednodušší.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lumír Návrat**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. května 2012

.....

Na tomto místě bych rád poděkoval všem, kteří mi pomohli tento projekt dotáhnout do konce, především Ing. Lumíru Návratovi, za jeho ochotu a přívětivý přístup.

Abstrakt

Cílem této práce je provedení analýzy možností synchronizace mobilních zařízení postavených na platformách Microsoft Windows Mobile 6.5 a Microsoft Windows Phone 7 se službou Google Calendar, popis a poměření již existujících softwarových řešení této problematiky a konečně návrh a implementace vlastního řešení pro platformu Microsoft Windows Mobile 6.5. Výsledná aplikace bude podporovat obousměrnou synchronizaci, hromadnou synchronizaci s více kalendáří v rámci jednoho uživatelského účtu a bude podporovat také synchronizaci úkolů (Google Tasks).

Klíčová slova: Microsoft, Windows Mobile, Windows Phone, Google, kalendář, úkol, mobilní aplikace, REST, API, synchronizace

Abstract

The goal of this thesis is to analyze the options of synchronization of the Microsoft Windows Mobile 6.5 and Microsoft Windows Phone 7 devices with the Google Calendar service, to describe and compare the already existing software solutions, and finally, to design and implement my own solution for the Microsoft Windows Mobile 6.5 platform. The resulting application will be capable of bi-directional synchronization, batch synchronization with multiple calendars within a single Google account, and will also support Google Tasks synchronization.

Keywords: Microsoft, Windows Mobile, Windows Phone, Google, calendar, task, mobile application, REST, API, synchronization

Seznam použitých zkratek a symbolů

API	–	Application programming interface
CF	–	Compact framework
GCal	–	Google Calendar
GTasks	–	Google Tasks
HTTP	–	Hypertext transfer protocol
JSON	–	JavaScript object notation
REST	–	Representational state transfer
MS	–	Microsoft
OS	–	Operating system
POOM	–	PocketOutlook Object Model
SDK	–	Software development kit
WinForms	–	Windows Forms
WM	–	Windows Mobile
XML	–	Extensible markup language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Kalendáře, diáře, a organizéry jsou užitečnými a nezřídka každodenními pomocníky v našich životech. Možnost efektivně spravovat svůj časový rozvrh a mít přehled nad blízkou i vzdálenější budoucností je zvláště v dnešní uspěchané době pro některé vyslovená nezbytnost, pro jiné třeba jen věc, na kterou si prostě zvykli a usnadňuje jim život. Ovšem bez ohledu na to, jakým způsobem tyto prostředky používají, či do jaké míry jsou na nich pracovně nebo jinak závislí, vždy je podstatný jeden fakt - aby měli ve všech těchto nástrojích konzistentní údaje.

V dobách dřívějších by to v praxi znamenalo ruční přepisování potřebných záznamů z kalendáře do diáře a naopak. Avšak dnes již máme k dispozici prostředky, které nás této námahy mohou zbavit. Většinu dat totiž běžně ukládáme (nebo máme možnost ukládat) v elektronické podobě a nejinak je tomu u diářů a kalendářů - a to přináší, za předpokladu odpovídající úrovně našeho vybavení, možnost synchronizace, neboli procesu zaručení konzistence dat mezi jednotlivými zařízeními či službami.

Jednou z takových služeb je velmi populární Google Calendar (aka GCal), jednoduchá a zároveň na funkce bohatá, webová aplikace ovládaná přímo v internetovém prohlížeči. Krom webového rozhraní je k ní ovšem možno přistupovat i přes Google Calendar API, což programátorům umožňuje vyvíjet synchronizační nástroje pro kalendář bez ohledu na použitou platformu.

V rámci této práce se budeme zabývat možnostmi Google Calendar synchronizace s mobilními platformami vytvořenými společností Microsoft, konkrétně Windows Mobile 6.5 a Windows Phone 7. Nejprve prozkoumáme podobu relevantních služeb a technologií společnosti Google, poté provedeme rozbor obou mobilních Windows platforem. Následovat bude analýza aktuálně dostupných synchronizačních řešení pro uvedené operační systémy a konečně návrh a implementace řešení vlastního (pro platformu Windows Mobile 6.5), ve kterém se také pokusíme eliminovat či zredukovat případné nedostatky konkurenčních aplikací.

2 Analýza služeb a technologií Google

2.1 Google Calendar a Google Tasks

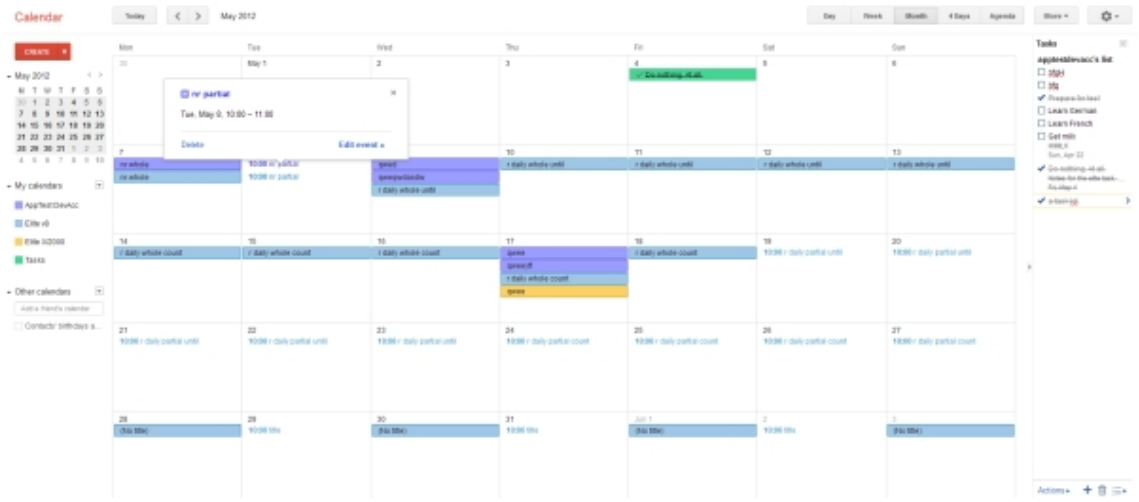
Hlavní entitou, kolem níž se otáčí tato práce v celém svém rozsahu, jak v teoretické tak praktické části, je služba Google Calendar. Je zařazena do široké škály služeb zdarma poskytovaných pod záštitou společnosti Google (přesněji Google Inc.) a jak je z názvu patrné, jejím primárním účelem je správa virtuálního kalendáře, resp. kalendářů. Náhled hlavního okna aplikace v režimu zobrazení měsíce, včetně panelu s úkoly u pravého okraje, je k vidění na obr. 1.

Mezi hlavní funkce služby patří:

- Správa jak celodenních (či vícedenních) tak krátkodobých událostí
- Rekurentní (opakované) události s bohatými možnostmi nastavení opakovacího vzoru
- Upozornění na událost (e-mailem či pop-upem)
- Správa více kalendářů v rámci jednoho Google účtu
- Správa úkolů - Google Tasks, podslužba kalendáře
- Měsíční, týdenní, denní a agendový režim zobrazení

Kalendář dále disponuje velmi užitečnou sociální fukcionalitou v podobě jeho zpřístupnění ostatním uživatelům. Mimojiné jej lze použít také jako formulář, kde se určitě události jeví uživatelům jako časové sloty, do kterých se můžou registrovat (funkce přihlášky). Tato inovativní čast funkčnosti služby by si jistě zasloužila důkladnější popis, avšak nemá nic společného se synchronizací (přinejmenším ne tou, která je předmětem této práce) a nebudeme se jí tedy dále zabývat.

Funkce tzv. to-do listů, neboli úkolů, je sice součástí kalendářové aplikace, ovšem má vlastní separátní API nazvanou Google Tasks a proto je také takto pracovně nazývána.



Obrázek 1: Google Calendar a Google Tasks ve webovém prohlížeči

2.2 OAuth 2.0

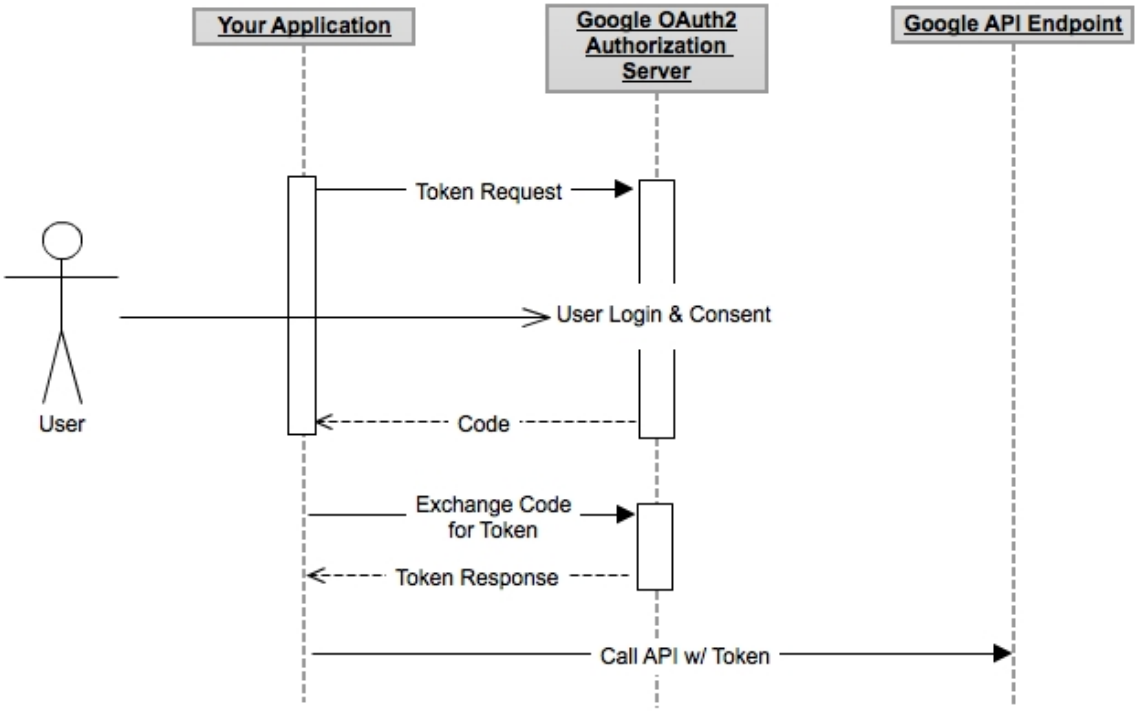
OAuth (zkratkový výraz pro Open Auth neboli Open Authorization) je protokol obstarávající otevřenou autorizaci uživatelů.

Otevřená autorizace obecně znamená, že do klientské aplikace uživatel nikdy nevkládá své autorizační údaje - místo toho je vloží mimo aplikaci do dialogu otevřené autorizační služby (typicky má podobu webového přihlášení zabezpečeného technologií SSL) a aplikace poté ze služby získá pouze určitý autorizační kód. Uživatel je tak chráněn před nebezpečnými aplikacemi, které by jeho přihlašovací údaje mohly zneužít.

Nejinak je tomu u technologie OAuth 2.0, což je momentálně primární, Googlem doporučený způsob autorizace pro přístup do Google APIs. Pomocí OAuth lze autorizovat přístup do veškerých API, které Google poskytuje [1].

Autorizační proces OAuth 2.0 pro typ aplikace *Installed application*, mezi které se řadí i aplikace vznikající v rámci této práce, je znázorněn sekvenčním diagramem na obr. 2 (oficiální diagram poskytovaný Google dokumentací) a vypadá následovně:

1. Uživatel si přeje provést přihlášení a klikne na příslušný ovládací prvek v aplikaci
2. Spustí se webový browser s Google webovou stránkou obsahující zabezpečený přihlašovací dialog
3. Uživatel se přihlásí
4. Uživateli je prezentován seznam oblastí, ke kterým aplikace požaduje přístup (tzv. OAuth scopes), např. kalendáře a úkoly
5. Uživatel potvrdí přístup (v opačném případě přihlašovací proces končí neúspěchem)



Obrázek 2: Sekvenční diagram autentizace pomocí OAuth 2.0

6. Uživateli je zobrazen autorizační kód
7. Uživatel autorizační kód zkopíruje do schránky, vloží jej zpět do aplikace a potvrdí
8. Aplikace odešle na autorizační server požadavek obsahující autorizační kód
9. Pokud je autorizace úspěšná, odpověd na požadavek obsahuje:
    - access_token - tento přístupový token je následně přikládán ke každému požadavku o přístup do API. Token je časově omezen na 3600s.
    - refresh_token - tento token se používá k získání nového access tokenu. Refresh token lze používat časově neomezeně až do chvíle, kdy uživatel odebere aplikaci přístupová práva v nastavení svého Google účtu.
    - epires_in - počet sekund, za kolik obdržený access token vyprší.
    - token_type - momentálně je hodnota toho parametru vždy "Bearer"

Tyto autorizační informace jsou uloženy, přihlašovací proces je dokončen a aplikace je připravena komunikovat s vlastními Google API.

2.3 Google APIs

Google APIs (celý oficiální název zní Google Apps Application APIs) jsou rozhraním umožňujím vývojařům aplikací přístupovat k naprosté většině Google služeb. Google jich momentálně skrze své API zpřístupňuje 36, prostou Translator API počínaje a Google+ API konče. Aplikace používají pro komunikaci s aktuálními API standardní protokol HTTP (resp. jeho šifrovanou SSL verzi, HTTPS) s využitím REST architektury (popsána níže) [2, 3].

Starší API se jmenují Google Data API (většinou jsou označovány jen zkráceně, GData) a jsou označené jako *deprecated* [4], tedy zastaralé nebo vyřazené. Jsou ovšem nadále funkční kvůli vyřazovací politice Googlu (deprecation policy), avšek jejich dokumentace je v dosti nekonzistentním stavu. GData API namísto RESTu používají publikační protokol ATOM. Ve své aplikaci jsem se těmto zastaralým a překonaným API samozřejmě vyhnul a dále se jim nebudu věnovat - zmiňuji je především proto, že se ještě objeví v analýze existujících řešení synchronizační problematiky.

Google APIs rovněž nabízejí klientské knihovny pro mnoho programovacích jazyků a platforem, vč. .NETu. Tyto knihovny (dostupné jak ve zkompilované podobě tak i jako zdrojový kód) osvobozují vývojaře aplikace od nutnosti implementovat vlastní přístup k API. Já ovšem předvytvořené klientské knihovny v této práci nevyužívám a ani nemohu, nejsou totiž podporovány v Compact verzi .NET Frameworku.

REST (Representational state transfer) není, i když je tak často mylně prezentován, komunikačním protokolem, ale pouze architekturou. REST používá model klient-server. Server zpřístupňuje své prostředky (prostředkem (resource) může být virtuálně jakýkoliv smysl dávající koncept, typicky nějaký objekt jako například událost v kalendáři) na základě požadávků klienta, přičemž zpět ke klientovi se přenáší *reprezentace* prostředku, vetšinou serializovaná forma objektu ve vhodném formátu [5].

Nejlepším znázorněním je příklad z praxe, kdy prostředkem je třeba již zmíněná událost v kalendáři, a její reprezentací se rozumí řetezec ve formátu JSON nebo XML, obsahující její serializaci. Tento řetezec je pak přenesen, vetšinou v HTTP odpovědi, ke klientovi a tam je (dle potřeby) deserializován.

Webové služby podporující REST (označované jako RESTful služby) typicky implementují následující HTTP metody:

1. GET odeslán na identifikátor..
   - ..prostředku: vrátí reprezentaci prostředku
   - ..kolekce prostředků: vrátí seznam prostředků v kolekci
2. DELETE odeslán na identifikátor..
   - ..prostředku: smaže prostředek
   - ..kolekce prostředků: smaže všechny prostředky v kolekci
3. POST odeslán na identifikátor..
   - ..kolekce prostředků: vloží nový prostředek do kolekce
4. PUT odeslán na identifikátor..
   - ..prostředku: nahradí prostředek
   - ..kolekce prostředků: nahradí celou kolekci prostředků jinou kolekcí

Ovšem konečná implementace (tedy podpora metod) záléží na konkrétní službě a prostředku. Může být podporováno více metod nebo naopak nemusí být implementovány některé z výše zmíněných, případně může být jejich funkce i pozměněna. Tento seznam je pouze orientační a jeho účelem je obecné znázornění funkce RESTu a RESTful webové služby.

2.3.1 Google Calendar REST API

Z Google Calendar API používám pro synchronizaci tyto čtyři typy prostředků:

- **CalendarList** - Seznam všech kalendářů v Google účtu. Tento získávám (GET) proto, aby uživatel mohl vybrat, které kalendáře chce synchronizovat.
- **CalendarListEntry** - Reprezentuje jeden kalendář. Tento prostředek nikdy samostatně přes REST nepřenáším, vždy získám jen jeho kolekci v podobě CalendarList.
- **Events** - Seznam všech událostí v kalendáři. Získávám jej (GET) na začátku synchronizačního procesu. Posílám na něj (POST) nové prostředky Event, pokud si to synchronizace vyžaduje.
- **Event** - Reprezentuje jednu událost. V RESTu použito při mazaní neplatných událostí (DELETE).

Kompletní popis těchto prostředků je obsažen v příloze B.

2.3.2 Google Tasks REST API

Z Google Tasks API používám pro synchronizaci tyto čtyři typy prostředků:

- **TaskLists** - Seznam všech seznamů úkolů v Google účtu. Tento získávám (GET) proto, aby uživatel mohl vybrat, které seznamy úkolů chce synchronizovat.
- **TaskList** - Reprezentuje jeden seznam úkolů. Tento prostředek nikdy samostatně přes REST nepřenáším, vždy získám jen jeho kolekci v podobě TaskLists.
- **Tasks** - Seznam všech úkolů v seznamu úkolů. Získávám jej (GET) na začátku synchronizačního procesu. Posílám na něj (POST) nové prostředky Task, pokud si to synchronizace vyžaduje.
- **Task** - Reprezentuje jeden úkol. V RESTu použito při mazaní neplatných úkolů (DELETE).

Kompletní popis těchto prostředků je obsažen v příloze C.

2.3.3 Aktivace a používání Google APIs

Google APIs jsou sice přístupné bezplatně (přesněji řečeno většina z nich), ale aplikaci používající API nelze "jen tak"začít vytvářet. Tomu musí předcházet (naštěstí poměrně jednoduchý) proces, vypadající následovně:

1. Je nutno vlastnit Google účet. Pokud jej vývojař nevlastní, musí jej vytvořit.
2. Dále je potřeba (bezplatně) registrovat projekt. K tomu slouží webové rozhraní pojmenované API Console. Jde v podstatě o administrační rozhraní projektů využívajících Google API.
3. Po registraci projektu je třeba aktivovat potřebné API (taktéž v API Console). V případě, že chce vývojař využívat pouze volně přístupné API, aktivace je otázkou jediného kliknutí na každou API, o kterou má zájem.
4. Posledním krokem je vytvoření autorizačních klíčů. Ty jsou dvojího typu:
   - **Simple API access** - tento klíč slouží k jednoduchému přístupu k API pokud aplikace nebude manipulovat s uživatelskými daty
   - **OAuth client ID** - tyto klíče jsou určeny pro OAuth autorizační process, který se přiřozeně používá, pokud je přístup k uživatelským datům potřeba

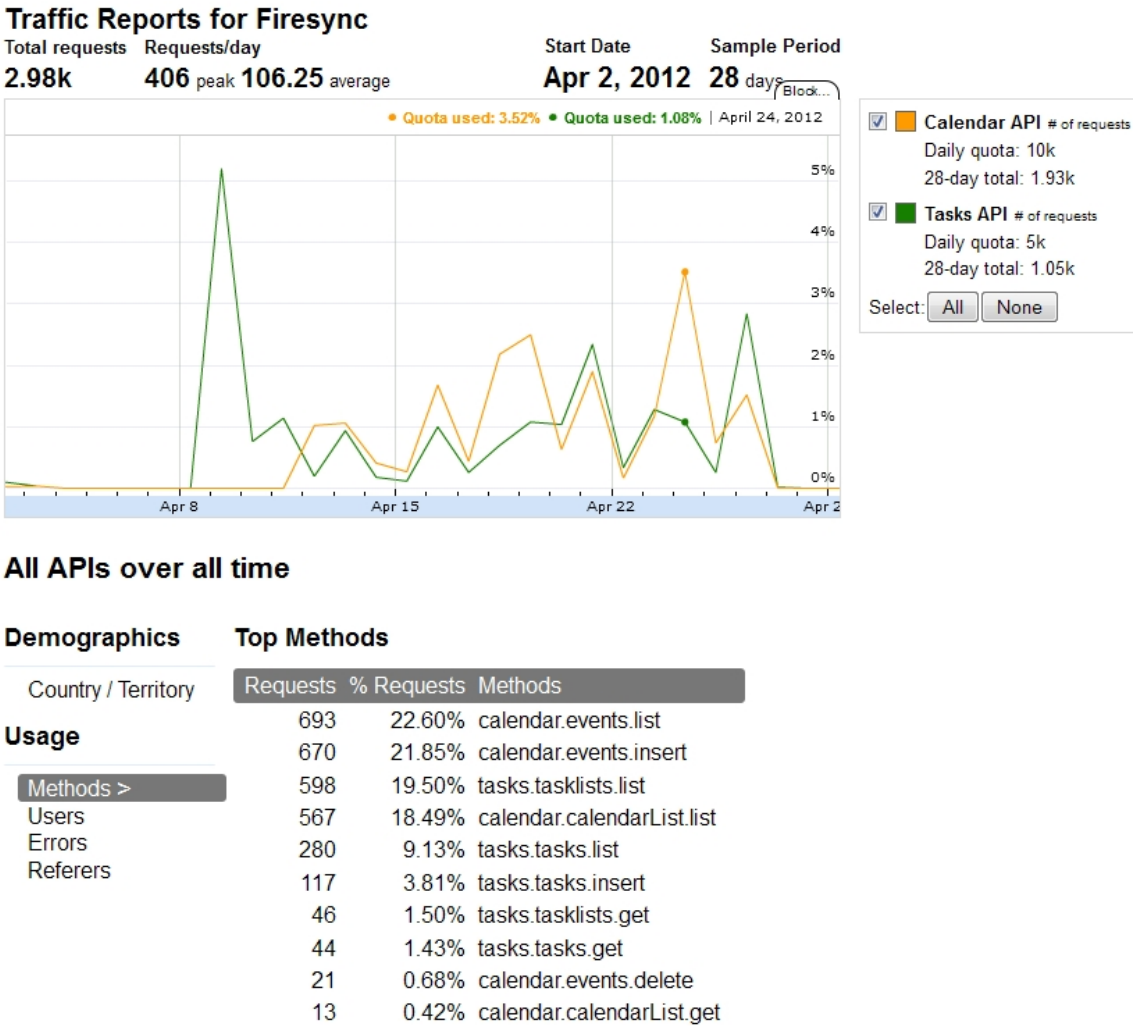
   V tomto systému se momentálně nachází bug. Dle všech dostupných oficiálních informací není Simple API access klíč vůbec potřeba, pokud je v aplikaci použita OAuth autorizace. Pokud ovšem tento klíč smažete, tak (přestože, vskutku, v aplikaci není ani vložen a je použita pouze OAuth autorizace) budou poté řádně autorizované požadavky na API odmítány chybovou odpovědí 403 Forbidden.

API console také umožňuje nastavit limity používání jednotlivých API. Tyto limity jsou dvojího typu:

- Limit počtu přístupů od jednoho uživatele za sekundu. Tento lze zdarma libovolně měnit.
- Denní limit celkového počtu přístupů všech uživatelů. Tento lze navýšit za poplatek. Výchozí hodnota poskytovaná zdarma (tzv. *courtesy limit*) se pohybuje řádově od sta k miliónu, podle API. V případě Google Calendar a Google Tasks API to je 10 tisíc, resp. 5 tisíc požadavků za den.

Pro svou aplikaci jsem musel navýšit výchozí limit počtu přístupů za sekundu (který se rovná pěti), protože při rychlém připojení a potřebě posílání většího množství nesynchronizovaných událostí či úkolů na servery Google nebyl problém tento limit překročit, což vedlo k přerušení připojení.

API console rovněž umožňuje monitorovat využití jednotlivých API v projektu, pomocí přehledného grafu (viz obr. 3).



Obrázek 3: Zobrazení využití API v Google API Console

3 Analýza mobilních platforem Microsoft Windows

Jedním z nejpodstatnějších produktů, kterými společnost Microsoft kdy obohatila svět informačních technologií, je platforma .NET Framework společně s programovacím jazykem C#. Tyto se používají nejen k vývoji různých typů aplikací pro desktopové operační systémy Windows, ale i pro jejich mobilní edice - v kompaktní verzi, přilehavě pojmenované .NET Compact Framework (.NET CF) [6].

S trochou nadsázky lze říci, že podpora .NETu a symbol okna jsou jediné věci, které mají systémy Windows Mobile a Windows Phone společné. Vskutku - Windows Phone je zatím s přehledem nejvýraznější změnou na poli mobilních OS od Microsoftu. Ze systému dříve určeného ryze pro enterprise segment se stal systém pro masy.

O rozdílech mezi WinMobile a WinPhone by se dala napsat celá nová kvalifikační práce, já se pochopitelně budu věnovat pouze rozdílům pro nás podstatným, tedy těm týkajícím se vývoje aplikací a synchronizace kalendáře a úkolů.

3.1 MS Windows Mobile 6.5

Systém podporuje .NET Compact Framework v poslední oficiální verzi, tedy 3.5, a uživatelské rozhraní aplikací je postaveno na grafické API Windows Forms. Vývoj software pro tuto platformu se tedy velmi podobá vývoji desktopových aplikací Windows Forms [7].

.NET CF vypadá přesně tak, jak by se podle názvu dalo očekávat - jde o odlehčenou verzi plnohodnotného frameworku, ochuzenou především o ”komfortní”prvky, tedy funkcionalitu, která programátorovi život ulehčuje, ale obejde se bez ní. Jako příklad uvedu absenci schopnosti ovládacího prvku Label měnit svou velikost na základě obsahu (tzv. funkce Autoresize). Tato funkčnost vývojaři může chybět, ale její manuální implementace (ideálně pomocí extension metody) je otázkou minuty.

Bohužel se může stát, že narazíme na absenci poněkud zásadnějšího prvku, často i celé třídy či namespace, kde by manuální implementace byla časově příliš náročná nebo z jiných důvodů nevhodná - v takovém případě nezbývá než se pokusit najít řešení pomocí jiné třídy, případně zamýšlenou funkčnost upravit tak, aby se v CF dala implementovat.

Pro přístup k událostem kalendáře a uloženým úkolům se používá API nazvaná PocketOutlook. Debugování aplikace při použití této API je poněkud obtížné, protože (aspoň z mé zkušenosti) vyvolává pouze dva typy vyjímek a obě jsou obecné. Při odstraňování problému je tedy potřeba řídit se programátorským instinktem a také spoléhat na internetové zdroje, případně aplikovat metodu pokus-omyl.

Kalendář ve Windows Mobile v sobě skýtá tři úskálí, všechna se přitom týkají rekurentních událostí.

- Asi nejzávažnejším je bug, kde rekurentní událost nastavená na týdenní opakování s periodou dvou nebo více týdnů, a zahrnující neděli plus minimálně jeden další den, bude generována nepřesně z toho důvodu, že je jako začátek týdne pro výpočet opakování vždy brána neděle, bez ohledu na to, že je v kalendáři jako začátek týdne nastaveno pondělí. Nejsem si vědom žádného řešení tohoto problému.
- Další problém nastává při rekurentním vzoru "Denně: v pracovní dny". Tato událost se totiž v API prezentuje nerozlišitelně od události s prostou denní rekurencí, přesná synchronizace proto není možná. Řešením je nahrazení této rekurence vzorem "Týdně: pondělí - pátek".
- Poslední problém je podobný předchozímu, opět se jedná o stav, kdy kalendář ukládá speciální informace o rekurenci do své vlastní databáze a ne přímo do objektu události (který je jako jediný dostupný přes API). Problém spočívá v tom, že když jako konec rekurence nastavíme pevné datum, dopočítá se počet opakování a vice-versa. Ovšem to, jakou variantu uživatel vybral, není možné z objektu události vyčíst. Má aplikace proto používá pro ukončení rekurence vždy variantu s pevným datem.

3.2 MS Windows Phone 7

Aplikace ve Windows Phone 7 jsou vyvíjeny na speciální verzi Microsoft Silverlight frameworku. Ta v případě WP7 sice běží na .NET Compact Frameworku 3.7, ovšem programátor s ním již nepracuje přímo, aplikaci staví jen na Silverlightu a ten o úroveň níže .NET CF využívá sám [8, 9].

Silverlight, stejně jako .NET CF 3.5, je taktéž odlehčenou verzí plnohodnotného .NET frameworku, zvláště pak Silverlight ve verzi pro Windows Phone. Oba frameworky ovšem zahrnují rozdílnou část jeho funkcionality a zjednodušeně se dá také říci, že Silverlight podporuje spíše jeho novější prvky než starší, v kontrastu s .NET CF.

Ovšem zcela nezásadnějším rozdílem oproti WM6.5 co se synchronizace týče je API pro přístup k uživatelským datům - aktuální verze Windows Phone SDK totiž umožňuje pouze read-only přístup ke kalendáři, a správu úkolů nepodporuje vůbec. Nicméně, já budu tuto skutečnost v rámci této práce ignorovat (také proto, že tento stav nemusí být trvalý) a mou WM6.5 aplikaci přesto navrhnu tak, aby její eventuální budoucí přenesení na WP7 bylo co možná nejjednodušší.

3.3 Vyvození důsledků ze srovnání platforem

- WM6.5 a WP7 používají naprosto odlišný framework uživatelského rozhraní, prioritou číslo jedna je proto důsledné oddělení uživatelského rozhraní od datových tříd naší aplikace, aby jeho nahrazení obnášelo co možná nejmenší (pokud možno nulové) změny ostatního zdrojového kódu.

- Některé třídy Compact Frameworku se ve WP7 nenachází, je proto vhodné dávat při vývoji aplikace pozor na použití těchto tříd a vyhnout se jim pokud je to možné. Výsledek tohoto snažení je shrnut v závěru práce.

4 Analýza stávajících synchronizačních řešení

K analýze jsem vybral tři existující synchronizační řešení pro OS Windows Mobile. Žádná řešení pro Windows Phone 7 jsem nenalezl, jsem si také vesměs jist, že žádná neexistují, z důvodů uvedených v analýze platformy výše a také proto, že WP7 (a především potom WP7.5) mají službu Google Calendar v sobě integrovánu.

4.1 PocketGCal

Velmi jednoduše vypadající (a fungující) synchronizační utilita od autora jménem Thom Shannon, k vidění na obr. 4. Charakteristika a výsledky analýzy:

- používá GData API
- přímá autorizace
- jednoduché rozhraní
- synchronizace pouze s primárním kalendářem účtu
- chybné zpracování časových zón
- rekurence ignorována

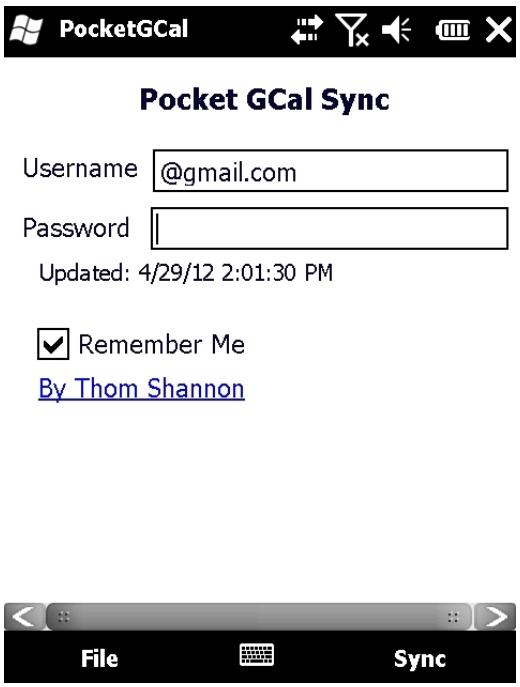
Po synchronizaci jsem sice objevil všechny události, které jsem měl, ovšem byly posunuty o dvě hodiny dopředu a jejich rekurenční vzory byly kompletně ignorovány.

4.2 GMobileSync

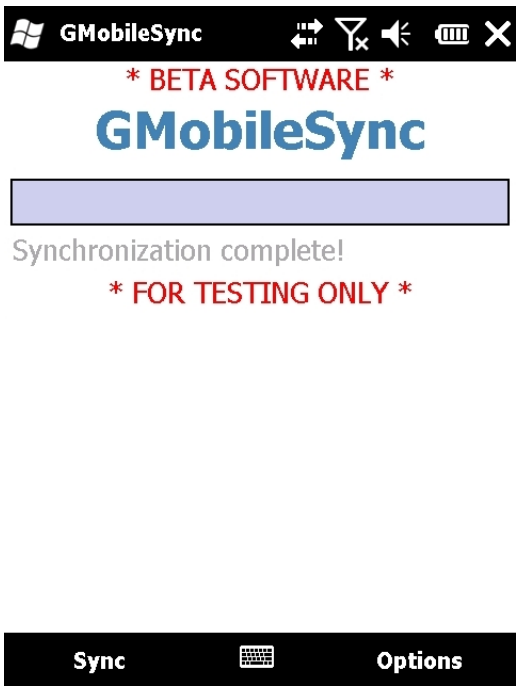
O něco důmyslnější řešení, zachycené na obr. 5, nabízí Eric Willis. Avšak přestože je toto již několikátá vydaná verze, stále má v hlavním okně jasně napsáno ”beta software”a ”for testing only”. A právem - analýza:

- používá GData API
- přímá autorizace
- synchronizace s více kalendáři bez možnosti volby
- pomalu reagující uživatelské rozhraní
- nekorektní chování aplikace

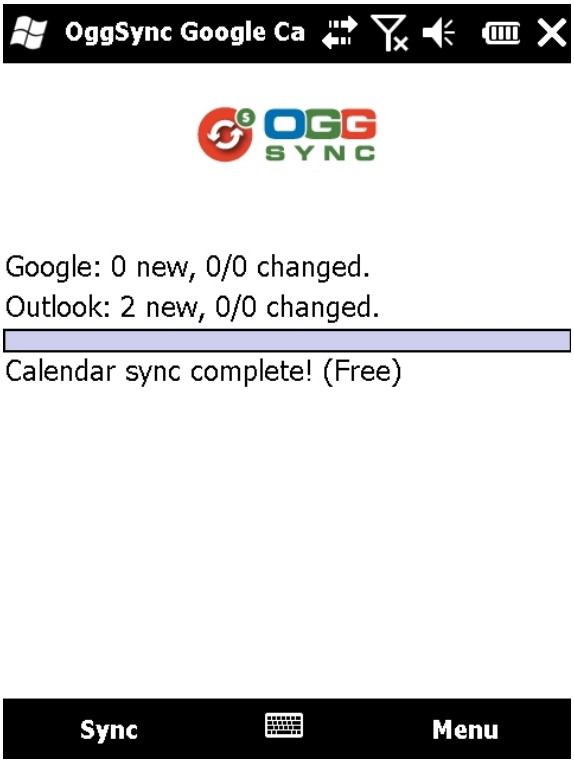
Aplikace se chová dosti nevypočitatelně. Občas provede bezchybnou synchronizaci (vč. rekurence), občas některé události vynechá, a občas je dokonce smaže bez zjevného důvodu.



Obrázek 4: PocketGCal



Obrázek 5: GMobileSync



Obrázek 6: OggSync

4.3 OggSync

Zdaleka nejprofesionálněji působící účastník, nabízí dokonce i placenou verzi. Ovšem ani toto řešení není bez chyby. K vidění je na obr. 6, analýza vypadá takto:

- používá GData API
- přímá autorizace
- synchronizace s více kalendáři s možností volby
- nekorektní chování aplikace

Testování vypadalo velmi nadějně, synchronizace oběma směry byla takřka perfektní, ale posléze se mi dvakrát stalo, že mi aplikace smazala veškeré události v přístroji a naopak jej zaplnila každodenní duplikací stejné události.

4.4 Vyvozené závěry

Nutno podotknout, že tato existující řešení nejsou zrovna velkou výzvou pro tvorbu konkurence - ani jedno nefunguje tak, jak by mělo. Jako možné příčiny se jeví (kromě implementačních chyb) zastaralost GData API a fakt, že aplikace byly designovány pro starší verzi Windows Mobile. Nicméně, z analýzy mohu odvodit tyto požadavky na mou aplikaci:

- **především** korektní a konzistentní funkčnost synchronizace samotné
- použití poslední verze Calendar API
- otevřená autorizace
- synchronizace s více kalendáři s možností volby
- svižné uživatelské rozhraní

5 Návrh vlastního řešení

Pro aplikaci jsem zvolil název **Firesync**. Vlastní logo a ikona aplikace jsou zachyceny na obrázcích 7 a 8.

5.1 Stanovení cílů implementace

Na základě zadání práce, popsaných analýz a také svých vlastních nároků na kvalitu a funkčnost jsem pro implementaci mého řešení stanovil následující požadavky:

- použití platformy Windows Mobile 6.5 s .NET Frameworkem ve verzi 3.5
- důsledné oddělení aplikačně-datového kódu od uživatelského rozhraní, aby bylo možné jeho případné pozdější nahrazení
- preference použití těch prvků .NET Compact Frameworku, které se nachází i ve Windows Phone verzi Silverlight frameworku
- implementace korektního a spolehlivého synchronizačního algoritmu
- funkce detekování a aplikování aktualizací událostí a úkolů
- uživatelské rozhraní:
  - založeno na Windows Forms
  - implementováno takovým způsobem, aby nemělo velkou odezvu
  - decentní vzhled, především respektující zavedené standardy barevného rozvržení
  - intuitivní používání
- využití posledních dostupných technologií tam, kde je to možné (především v případě Google API)
- implementace otevřené autorizace chránící uživatele
- schopnost synchronizovat přístroj oboustranně proti vybranému kalendáři
- schopnost synchronizovat přístroj jednostranně (ve směru z Googlu do přístroje) proti více kalendářům
- implementace ochrany proti time-outům pří síťové komunikaci
- perzistence uživatelských nastavení

Obrázek 7: Firesync logo

Obrázek 8: Firesync ikona

5.2 Návrh algoritmu použití aplikace

Použití aplikace k synchronizaci dat sestává z tří hlavních kroků:

- Autentizace uživatele
- Výběr Google kalendářů a seznamů úkolů
- Samotný synchronizační proces

Nyní se na jednotlivé kroky podíváme podrobněji.

5.2.1 Autentizace uživatele

Aplikace používá k autentizaci uživatele a autorizaci požadavků technologii OAuth 2.0. Finálním cílem autentizace uživatele je získání přístupového a obnovovacího tokenu, z čehož první jmenovaný se poté používá k autorizaci všech požadavků na Google API a je časově omezený (typicky platný na jednu hodinu), druhý pak pro získání nového přístupového tokenu.

Proces autentizace v aplikaci probíhá přesně dle scénáře definovaného v mém popisu technologie OAuth v sekci 2.2. Dochází k němu pouze pokud se jedná o první spuštění aplikace (případně pokud si uživatel přeje změnit přihlášený účet), v opačném případě

dojde pouze k získání uložených tokenů z pevného úložiště a případné (automatické) obnově přístupového tokenu, pokud jeho platnost již vypršela.

5.2.2 Výběr Google kalendářů a seznamů úkolů

V této fázi je již uživatel vždy autentizován, autorizační modul aplikace má k dispozici platná data a je připraven autorizovat vytvořené požadavky na Google API. Toho využijeme ihned na začátku tohoto kroku, kdy dojde ke stažení seznamu všech kalendářů a seznamů úkolů.

Seznamy jsou poté prezentovány uživateli v přehledné formě a ten následně vybírá ty prostředky, které má zájem synchronizovat. Vybere vždy jeden kalendář a jeden seznam úkolů pro obousměrnou synchronizaci a libovolný počet pro jednosměrnou synchronizaci.

Jakmile je uživatel s výběrem spokojen, tento krok ukončí kliknutím na příslušný ovládací prvek.

5.2.3 Samotný synchronizační proces

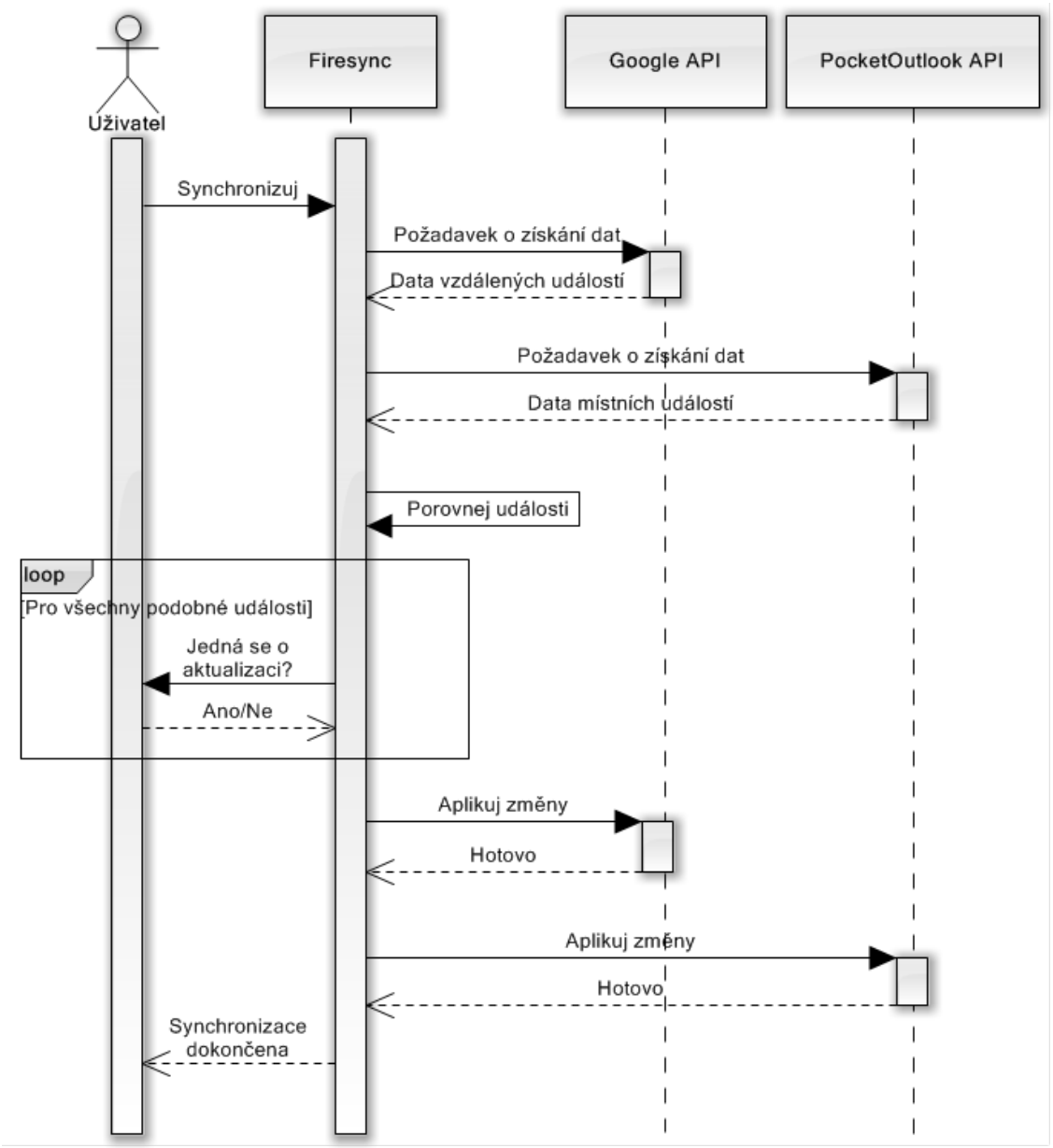
Algoritmus se zde liší podle toho, zda jde o synchronizaci obousměrnou nebo jednosměrnou. Je ovšem identický pro kalendáře i seznamy úkolů a popíšu jej tedy jen pro kalendář.

Obousměrná synchronizace
Sekvenční diagram popisující obousměrnou synchronizaci je k vidění na obr. 9.

1. **Načtení prostředků** - dochází ke stažení všech událostí příslušného kalendáře z Google serverů, dále k načtení všech událostí z přístroje a nakonec ke konverzi událostí přístroje do formátu Google událostí, což je nezbytné pro následující komparační proces.

2. **Komparace** - všechny vzdálené a místní události jsou postupně porovnány stylem "každý s každým", jejich míra podobnosti je číselně ohodnocena a v případě, že je nalezena vysoká podobnost[1] či shoda, je tato informace uložena v samotném objektu události. Pokud byla již událost jednou označena jako shodná (synchronizovaná), znovu již není s dalšími událostmi poměřována, což by bylo zbytečné plýtvání časem CPU.

[1] Potřebná míra podobnosti je nastavena uživatelem.



Obrázek 9: Sekvenční diagram obousměrné synchronizace

3. **Zpracování podobných událostí** - V případě, že si je nějaká dvojice místních a vzdálených událostí podobná (ale nejsou přitom zcela identické), může se jednat o aktualizaci události. Tato skutečnost je ohlášena uživateli, který dostává možnost vybrat, zda:

   - vzdálená událost je aktualizací místní - v tom případě bude místní událost smazána a vzdálená ji nahradí
   - místní událost je aktualizací vzdálené - vzdálená událost bude smazána a místní ji nahradí
   - nejde o aktualizaci

   V tomto kroku nedochází k samotnému provedení smazání a náhrady, pouze k uložení příslušné informace do synchronizačního parametru objektu události, stejně jako v komparačním kroku.

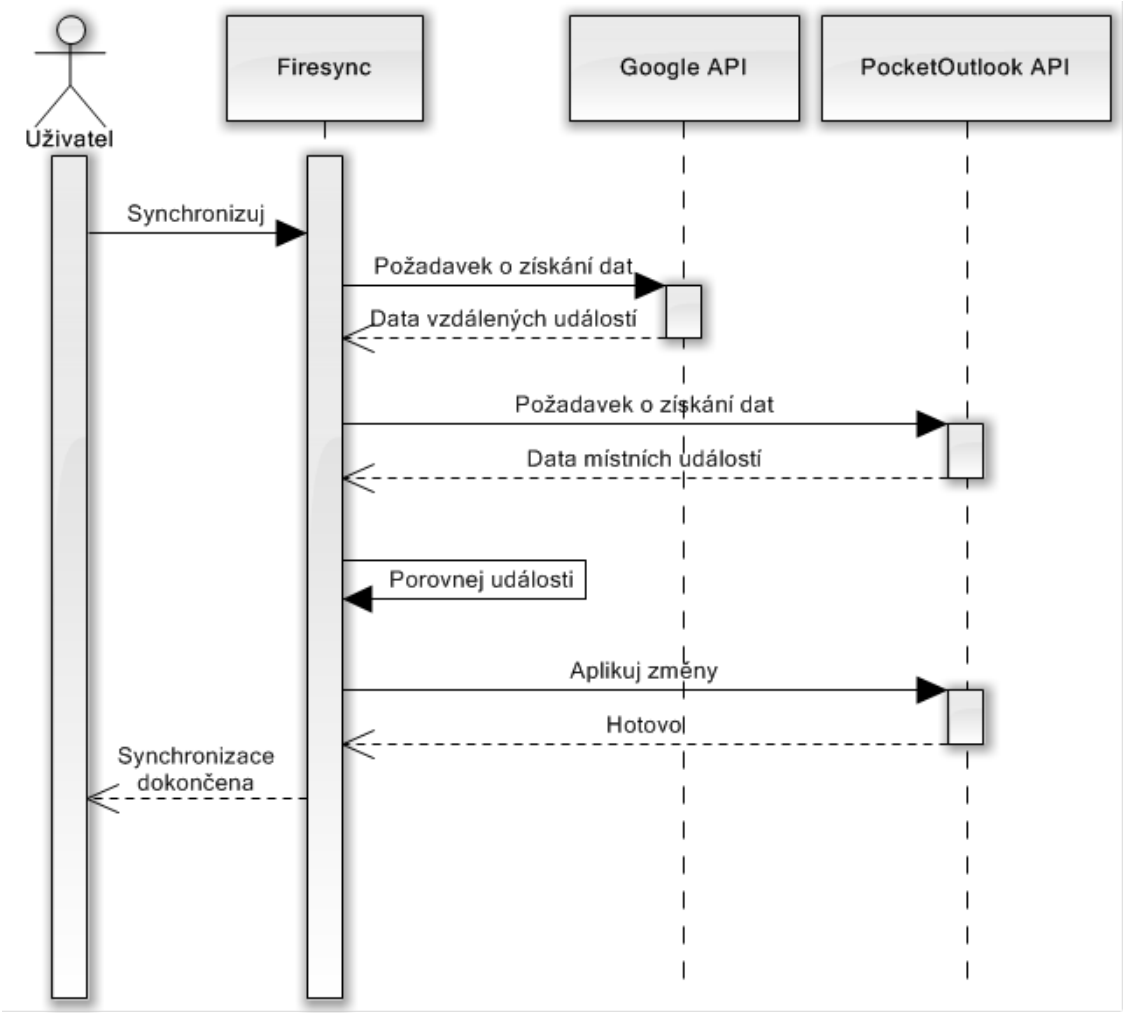
4. **Provedení změn** - všechny místní i vzdálené události jsou postupně zpracovány a na základě jejich přiřazených synchronizačních parametrů dochází k jejich uložení na protejší stranu (z přístroje na Google či naopak) pokud se jedná o událost novou, ke smazání pokud je událost zastaralá a již neplatná, případně k žádné akci pokud je událost již v synchronizovaném stavu.

Jednosměrná synchronizace z Googlu do přístroje
Algoritmus je velmi podobný obousměrné synchronizaci, v podstatě jen nezahrnuje zpracování podobnosti (roli zde hraje pouze absolutní shoda, možnost aktualizace je ignorována) a navíc poslední krok, kde dochází k uložení nových událostí, se týká pouze vzdálených událostí. Sekvenční diagram je zachycen na obr. 10.

1. **Načtení prostředků** - dochází ke stažení všech událostí příslušného kalendáře z Google serverů, dále k načtení všech událostí z přístroje a nakonec ke konverzi událostí přístroje do formátu Google událostí, což je nezbytné pro následující komparační proces

2. **Komparace** - všechny vzdálené a místní události jsou postupně porovnány stylem ”každý s každým”, jejich míra podobnosti je číselně ohodnocena a v případě, že je nalezena shoda, je tato informace uložena v samotném objektu události. Pokud byla již událost jednou označena jako shodná (synchronizovaná), znovu již není s dalšími událostmi poměřována, což by bylo zbytečné plýtvání časem CPU.

3. **Provedení změn** všechny vzdálené události jsou postupně zpracovány a na základě jejich přiřazených synchronizačních parametrů dochází k jejich uložení do přístroje, pokud se jedná o událost novou nebo k žádné akci pokud, je událost již v synchronizovaném stavu.



Obrázek 10: Sekvenční diagram jednosměrné synchronizace

6 Implementace

Aplikace je implementována, dle zadání, pro platformu Windows Mobile 6.5. Je napsána v jazyce C# a postavena na .NET Compact Frameworku ve verzi 3.5.

6.1 Použité nástroje a knihovny třetích stran

Pro vývoj aplikace byly použity následující nástroje:

- Vývojové prostředí Microsoft Visual Studio 2008 SP1

  jeho novější verze, VS2010, bohužel vývoj aplikací pro WM 6.5 nepodporuje, pouze vývoj aplikací pro Windows Phone.

- Windows Mobile 6 SDK

  přidává do VS2008 podporu pro vývoj WM6 aplikací (instalace VS2008 samotného zahrnuje pouze WM5 SDK), včetně emulátorových obrazů.

- Windows Mobile 6.5 DTK (Developer Toolkit)

  rozšiřuje WM6 SDK o podporu a emulátorové obrazy WM6.5. WM6 SDK je prerekvizitou pro instalaci.

- Microsoft Virtual PC 2007

  tento virtualizační nástroj sám o sobě sice není při vývoji aplikace použit, ovšem nainstaluje do systému síťový ovladač potřebný pro zprovoznění internetové konektivity v emulátorech. Tento ovladač dnes nelze oficiálně získat jinou cestou, než v rámci instalace Virtual PC 2007.

Aplikace využívá pro práci s formátem JSON knihovnu Json.NET 3.5 R8, vytvořenou a poskytovanou k volnému použití Jamesem Newton-Kingem.
Pro tvorbu obrázků a ikon byly použity tyto nástroje:

- PhotoFiltre 6.5.2

  středně pokročilý freewarový editor, autor Antonio Da Cruz

- GIMP 2.6.11

  GNU Image Manipulation Program, pokročilý open-source editor

- IrfanView 4.28

  freewarový nástroj pro jednoduché úpravy, autor Irfan Skiljan

- IconPro Icon Handler

  Jednoduchá utilita od Microsoftu sloužící k tvorbě ikon z obrázků

6.2 Rozvržení aplikace

Aplikace se skládá ze dvou assemblies. Těmi jsou FiresyncLib, což je třídní knihovna (.dll) a obsahuje aplikačně-datovou část kódu, a pak Firesync, spustitelná aplikace, ve které se nachází uživatelské rozhraní na bázi Windows Forms. Firesync samozřejmě referencuje FiresyncLib, čímž je zaručeno oddělení UI od funkčního kódu. FiresyncLib potom referencuje výše zmíněnou Json.NET knihovnu (pojmenovanou Newtonsoft.Json.Compact.dll).

Většina dat obou assemblies se nachází v namespace Firesync, výjimkou jsou objekty reprezentující prostředky služeb Google Calendar a Google Tasks a třídy pro přístup k nim, nácházející se v namespacech Firesync.GoogleCalendar, resp. Firesync.GoogleTasks.

6.3 Spuštění aplikace a autentizace

První provedenou akcí po spuštění aplikace Firesync je načtení uživatelského nastavení ze souboru *settings.xml*. Ze souboru se XML deserializací vytvoří instance třídy AppSettings a reference na ni se uloží do statické property třídy Global, která, jak název napovídá, slouží k zajištění dostupnosti objektů v rámci celé aplikace.

Tato třída také obsahuje několik statických metod, které se k objektům v ní obsaženým vztahují. Už samotné načtění nastavení ze souboru je realizováno metodou Global.LoadSettings(). V případě, že soubor s nastavením není dostupný, instance AppSettings se vytvoří s defaultními hodnotami.

Objekt typu AppSettings, který mimo nastavení obsahuje i statistiky použití aplikace, zahrnuje tyto prvky:

- TimesRun - celkový počet spuštění aplikace
- UserEmail - e-mailová adresa přihlášeného uživatele
- EventSimilarityTreshold - definuje hranici, od které se události klasifikují jako podobné
- TaskSimilarityTreshold - definuje hranici, od které se úkoly klasifikují jako podobné
- HttpRequestsMade - celkový počet odeslaných HTTP požadavků
- Timeouts - počet HTTP požadavků, u kterých vypršel časový limit odpovědi
- AverageLatency - průměrná doba odezvy při HTTP komunikaci, v milisekundách
- DefaultResponseStatus - při odeslání události na Google kalendář se v případě, že je sám uživatel uveden v seznamu hostů, nastaví jeho "reakce na pozvání"podle tohoto nastavení

Mimojiné se dále po startu aplikace vytvoří instance všech grafických objektů (především menu). Tyto objekty nejsou nikdy odstraněny, pouze schovávány a zobrazovány, čímž se vyhneme zpomalení při přechodech do nových menu. Reference na ně jsou udržovány ve statických properties třídy Window.

6.3.1 Autentizace

Tím končí obecná inicilizace aplikace a začíná proces autentizace. Autentizaci a autorizaci obstarává třída Auth, která zaprvé nese autorizační informace ve své instanci (tato instance je pak referencována třídou Global), a zadruhé poskytuje statické autentizační a autorizační metody.

Nejrpve dochází k pokusu o načtení bezpečnostních tokenů ze souboru metodou Auth.LoadTokens().

```
public static bool LoadTokens()
{
    if (! File . Exists(Global.AppDataDir + "\\tokens.xml"))
    {
        return false;
    }
    XmlSerializer xs = new XmlSerializer(typeof(Auth));
    using (TextReader reader = new StreamReader(Global.AppDataDir + "\\tokens.xml"))
    {
        Global.Authorization = (Auth)xs.Deserialize(reader);
    }
    return true;
}
```

Výpis 1: Metoda Auth.LoadTokens()

V případě úspěchu následuje kontrola platností přístupového tokenu pomocí metody ExpCheck() neboli expiration check.

```
public bool ExpCheck()
{
    if (expiry_time > DateTime.Now)
        return true;
    return false;
}
```

Výpis 2: Metoda Auth.ExpCheck()

expiry_time je private DateTime field instance Auth a nese v sobě datum a čas konce platnosti přístupového tokenu. Metoda tedy vrací **true** pokud čas vypršení ještě nebyl dosažen a token je stále platný.

Pokud i metoda ExpCheck ohlásí úspěch, proces autentizace je u konce. V opačném případě je access token potřeba obnovit pomocí refresh tokenu, k čemuž slouží metoda Auth.RenewToken():

```
public static void RenewToken()
{
    string query =
        " client_id =" + Uri.EscapeDataString(AppInfo.client_id) + "&" +
        " client_secret =" + Uri.EscapeDataString(AppInfo.client_secret) + "&" +
        "refresh_token=" + Uri.EscapeDataString(Global.Authorization.refresh_token) + "&" +
        "grant_type=" + Uri.EscapeDataString("refresh_token");

     RequestContainer rcon = Net.Post("https://accounts.google.com/o/oauth2/token", query,
         false, false, ContentType.Formurlencoded);

    Auth a = Utils.JDeserialize<Auth>(Net.GetResponse(rcon).Content);
    a.CalcTimestamp();
    Global.Authorization.access_token = a.access_token;
    Global.Authorization.expiry_time = a.expiry_time;

    StoreTokens();
}
```

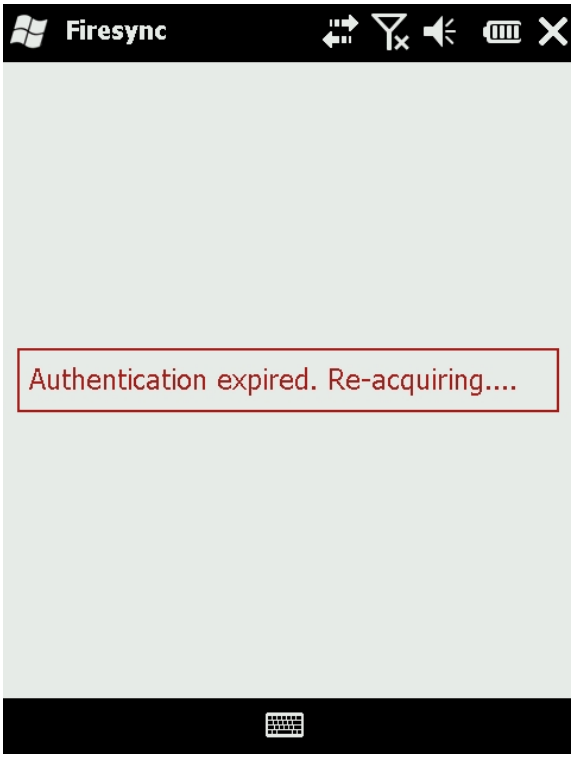
Výpis 3: Metoda Auth.RenewToken()

Na tuto metodu se podíváme blíže. Cílem metody je získat nový access token z odpovědi na požadavek HTTP POST, který musí metoda nejdříve vytvořit. Tělo požadavku je tvořeno řetězcem skládajícím se ze čtyř parametrů:

- **client_id a client_secret** jsou konstantní identifikátory unikátní pro každou aplikaci, jsou přirazeny při aktivaci OAuth autorizace v Google API Console.
- **refresh_token** odpovídá hodnotě načtené ze souboru
- **grant_type** má u tohoto požadavku vždy hodnotu "refresh_token"

Objekt typu RequestContainer obsahuje množinu identických HTTP požadavků a případný obsah, který má být do nich zapsán. Nahrazení jednoduchého HTTP požadavku tímto objektem má význam v implementované mechanice ošetřování timeoutů síťové komunikace, tato mechanika je popsána v sekci 6.4, stejně jako třída Net starající o veškerý síťový provoz aplikace. Pro tuto chvílí nám ovšem stačí zjednodušený popis, tedy že metoda Net.Post vrací objekt RequestContained obsahující HTTP POST požadavky odpovídající zadaným parametrům a Net.GetResponse zavolaná na tento objekt poté vrací odpověď na jeden z těchto (identických) požadavků.

Obsah odpovědi je ve formátu JSON a je tedy potřeba jej deserializovat pomocí metody Utils.JDeserialize. Třída Utils obsahuje množství statických metod, které obsahují procedury používáné v obecném chodu celé aplikace. Účelem třídy je tedy prostá prevence opakování stejného kódu na více místech. Obsahuje jak běžné statické metody tak i extension metody.



Obrázek 11: ProgressBox oznamující probíhající akci

Metoda JDeserialize<>tedy samozřejmě deserializuje řetězec ve formátu JSON na objekt zadaného typu. Na výsledném objektu typu Auth zavoláme metodu CalcTimeStamp(), která z hodnoty expires_in, tedy počet sekund, za kolik nový přístupový token vyprší, vypočítá konkrétní datum a čas a uloží je do vlastnosti expiry_time.

Poté již zbývá jen aktualizovat parametry globálního objektu Auth (celý jej nahradit nemůžeme, protože odpověď na obnovení přístupového tokenu pomocí obnovovacího tokenu neobsahuje samotný obnovovací token, a není tedy obsažen ani v našem lokálním Auth objektu). Nakonec globální autorizační objekt uložíme do souboru metodou StoreTokens() a proces obnovení je hotov.

Na to, že probíhá obnovení přístupového tokenu, je uživatel upozorněn pomocí objektu ProgressBox (obr. 11), který se napříč celou aplikací používá k informování uživatele o právě prováděné akci. Tento objekt je schopen provádět jednoduchou animaci v podobě přibývajících teček za zprávou, čímž je uživateli intuitivně naznačeno, že má vyčkat na dokončení akce.

6.3.2 Průběh OAuth 2.0 autentizace nového uživatele

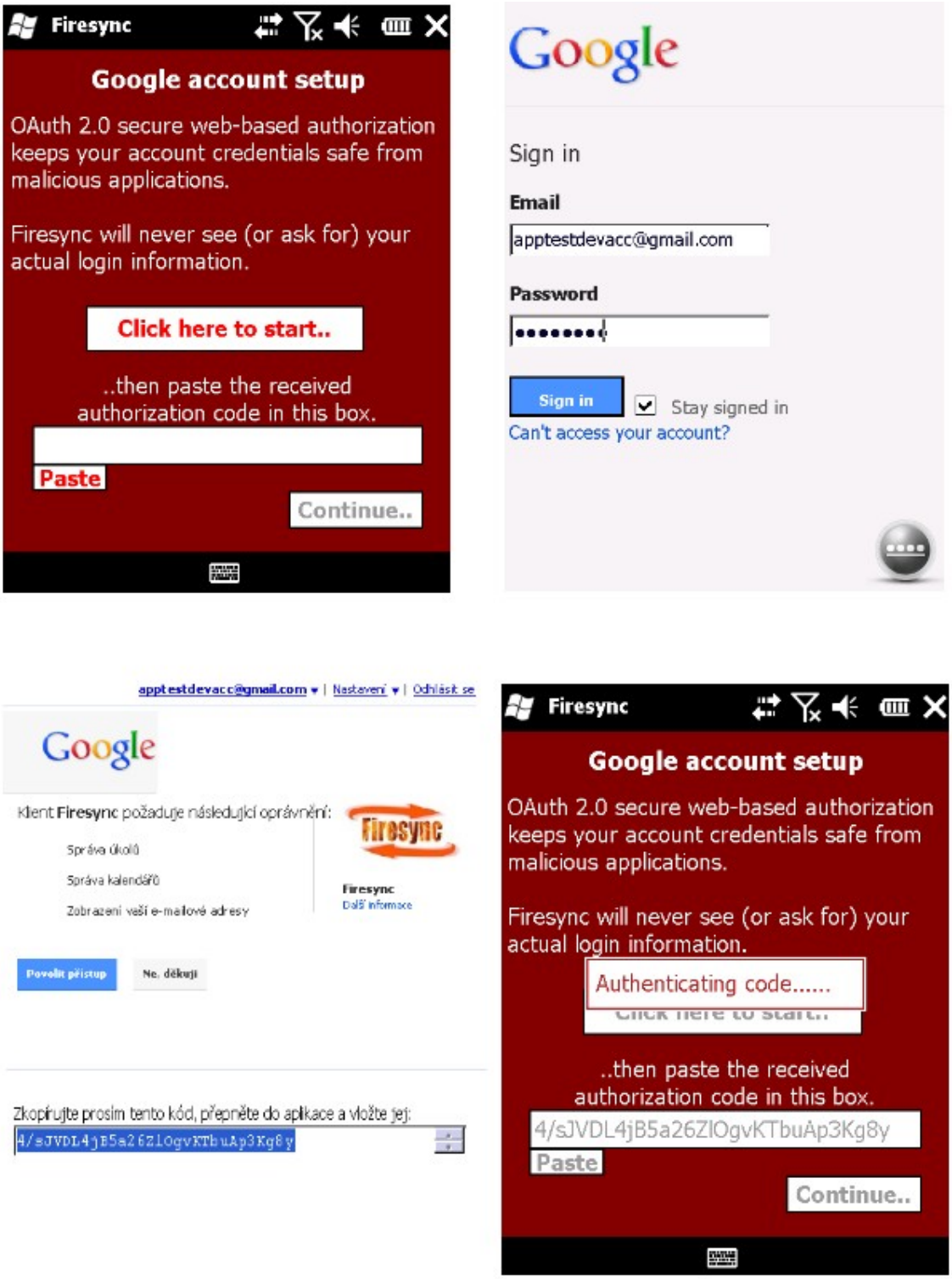
V případě, že se nepodaří načíst soubor s přístupovými tokeny, spustí se proces autentizace nového uživatele. Tento proces byl abstraktně popsán dříve v prezentaci technologie OAuth, my mu nyní dáme formu.
Průběh Firesync autentizace z pohledu uživatele je zachycen v pěti částech na obr. 12 a vypadá takto:

1. Úvodní obrazovka s instrukcemi
2. Po kliknutí na tlačítko se spustí webový prohlížeč a nabídne přihlášení
3. Po přihlášení následuje požadavek přístupu ke kalendářům a úkolům uživatele
4. Po odsouhlasení je vrácen autorizační kód
5. Kód je vložen zpět do aplikace a vyměnen na pozadí za přístupové tokeny

Uvnitř aplikace proces probíhá následovně:

- Po kliknutí na tlačítko "Click here to start.."je zavolána metoda Auth.WebAuth(). Ta obsahuje jediný příkaz, a to je volání metody Process.Start(string uri, ""), která zadaný webový odkaz automaticky spustí v defaultním webovém prohlížeči.
- Po návratu do aplikace, vložení kódu a kliknutí na tlačítko "Continue..", je zavolána metoda Auth.FetchTokens(string code). Tato metoda probíhá téměr identicky jako Auth.Renew, až na ten rozdíl, že požadavek obsahuje autorizační kód místo refresh tokenu a ten je naopak navíc obsažen v následné odpovědi. Tuto metodu si tedy popisovat nebudeme.

Úspěšným provedením metody FetchTokens autentizační proces končí.



Obrázek 12: Průběh OAuth 2.0 autentizace v aplikaci Firesync

6.4 Síťová komunikace

Veškerá síťová komunikace aplikace je realizována skrze metody statické třídy Net (nezaměňovat s namespace System.Net!). Nejprve ovšem zmíním podpůrné objekty RequestContainer a NetResponse, které se objevují na vstupech a výstupech většiny metod třídy Net.

RequestContainer nahrazuje jednoduchý objekt požadavku HttpWebRequest a v sobě obsahuje kolekci několika identických kopií tohoto objektu, společně s případným řetězcem, který má být k požadavku připojen jako jeho tělo.

Důvod pro použití RequestContaineru je prostý. Síťová komunikace přes internet není stoprocentně spolehlivá a čas od času se stane, že požadavek se nedočká odpovědi v potřebném čase. Tomuto jevu se říká timeout.

Timeout je nejčastěji řešen opakovaným odesláním stejného požadavku, a stejně tak je tomu v případě Firesyncu. Ovšem kopírování existujících požadavků není efektivním řešením získání duplikátu, lepší je čisté duplikáty vygenerovat ze vstupních dat okamžitě. A přesně k tomu slouží objekt RequestContainer, resp. k uchovávání těchto požadavků.

Objekt NetResponse se nachází na opačné straně. Do jeho instancí jsou ukládány odpovědi na webové požadavky. Obsahuje v sobě status kód odpovědi, slovní popis status kódu, tělo odpovědi pokud nějaké existuje, informaci, zda došlo k výjimce a v případě, že ano, tak i výjimku samotnou.

A nyní již samotná třída Net. Jádrem třídy jsou 4 metody pro vytváření objektů webových požadávků (resp. RequestContainerů), jedna metoda pro každý z typů HTTP metod GET, POST, PUT a DELETE, a dále metoda GetResponse, která pro vytvořený RequestContainer získá odpověď a vrátí ji ve formě NetResponse. Dohromady tyto metody poskytují pohodlné rozhraní pro tvorbu libovolných webových požadavků a zpracování jejich odpovědí.

Zmíněné čtyři metody jsou si navzájem velmi podobné, a proto si popíšeme jen jednu, konrétně Post.

```
public static RequestContainer Post(string uri, string content, bool keepalive, bool
    appendAuth, ContentType ctype)
{
    RequestContainer rcon = new RequestContainer(content);
    for (int i = 0; i <= resend; i++)
    {
        HttpWebRequest req = (HttpWebRequest)WebRequest.Create(uri);

        req.Method = "POST";
        if (appendAuth)
            Global.Authorization.AppendAuthHeaders(req);
        switch (ctype)
        {
            case ContentType.Formurlencoded:
                req.ContentType = "application/x-www-form-urlencoded";
                break;
            case ContentType.Json:
                req.ContentType = "application/json";
                break;
        }
        req.KeepAlive = keepalive;
        req.ContentLength = content.Length;

        rcon.Requests.Add(req);
    }

    return rcon;
}
```

Výpis 4: Metoda Net.Post

Vstupní parametry:

- uri - cílová adresa požadavku
- content - obsah těla požadavku
- keepalive - určuje, zda má být připojení perzistentní
- appendAuth - určuje, zda má být k požadavku připojen autorizační header obsahující aktuální access token v objektu Global.Authorization (instance typu Auth)
- ctype - content type, typ obsahu těla požadavku

Nejrpve dochází k vytvoření instance objektu RequestContainer. Do kontejneru jsou poté přidány identické kopie požadavku definovaného vstupními parametry, a to v celkovém počtu *resend*+1, kde resend je konstanta třídy Net, v době psaní tohoto textu nastavena na hodnotu 2 - tedy každý RequestContainer obsahuje tři identické požadavky.

Proces vytvoření požadavků nepotřebuje komentář, pouze si ukážeme zmíněnou jednoduchou metodu Auth.AppendAuthHeaders, která požadavek autorizuje.

```
public void AppendAuthHeaders(HttpWebRequest request)
{
    if (!ExpCheck())
        Auth.RenewToken();
    request.Headers.Add("Authorization", _token_type + " " + _access_token);
}
```

Výpis 5: Metoda Auth.AppendAuthHeaders

Pokud v době žádosti o připojení autorizace k HTTP požadavku není přístupový token již platný, před připojením headeru dojde k jeho aktualizaci.

Metoda GetResponse (zdrojový kód se kvůli délce nachází až na další stránce) má jediný vstupní argument, a tím je produkt jedné ze zmíněných čtyř metod, tedy objekt RequestContainer.

Kolekce požadavků v objektu je postupně iterována - v ideálním stavu pouze jednou, ale v případě potřeby (tj. když dochází k timeoutům) jsou posílany další requesty tak dlouho, dokud nějaké zbývají. Prakticky se tedy (při aktuálním nastavení) jedná o jeden request standarní a dva záložní (leč identické) pro případ výskytu timeoutů.

Pokud v kontejneru dojdou požadavky (tzn. pokud všechny požadavky skončí timeoutem, protože ostatní typy výjimek k další iteraci nevedou) tak je manuálně vyvolána výjimka na tento stav upozorňující. Pokud je ovšem kterýkoliv z požadavků úspěšně proveden, odpověď je roztříděna do objektu NetResponse a metoda jej ihned vrací (tzn. k dalším iteracím kontejneru již nedochází).

Ještě poskytnu komentář k metodě HttpWebRequest.GetRequestStream(), která se v popisované metodě Net.GetResponse nachází ihned nad voláním HttpWebRequest.GetResponse(). Mnoho programátorů (troufl bych si dokonce říci, že naprostá většina) se mylně domnívá, že požadavek je vždy odeslán až voláním metody HttpWebRequest.GetResponse(). Tak je tomu vskutku u požadavků, které neobsahují tělo (jako je GET a DELETE). Skutečnost je ovšem taková, že nevinně vypadající metoda HttpWebRequest.GetRequestStream(), která na první pohled pouze vkládá tělo do lokálního objektu požadavku, jej odesílá. Proto je praktické tuto metodu v procedurách, které pracují s oběma typy požadavků (s tělem i bez těla) umístit okamžitě před HttpWebRequest.GetResponse(), čímž se zajistí konzistentní provedení procedury bez ohledu na to, zda požadavek tělo má, či ne.

Já jsem tento fakt bohužel objevil poměrně bolestivým způsobem, když jsem v implementaci ochrany proti timeoutům zapisoval do všech duplicitných požadavků v kontejneru jejich požadované tělo ihned - tzn. GetRequestStream jsem volal již v metodě Net.Post, a to celkem třikrát. Velmi dlouho mi trvalo přijít na to, proč se ne-timeoutující požadavky odesílají třikrát, tím spíše když je metoda HttpWebRequest.GetResponse() volána pouze jednou.

```
public static NetResponse GetResponse(RequestContainer rcon)
        {
            if (rcon == null || rcon.Requests == null || rcon.Requests.Count == 0)
                throw new Exception("Received empty or null HTTP request list.");
            foreach (HttpWebRequest request in rcon.Requests)
            {
                NetResponse response = new NetResponse();
                Stopwatch sw = new Stopwatch();
                try
                {
                    request.Timeout = timeout;
                    Global.Settings.HttpRequestsMade++;

                    sw.Start();
                    if (rcon.ContentPresent)
                        Utils.StringToStream(request.GetRequestStream(), rcon.Content);
                    HttpWebResponse r = (HttpWebResponse)request.GetResponse();
                    sw.Stop();
                    Global.Settings.AddLatency((int)sw.ElapsedMilliseconds);
                    response.Statuscode = (int)r.StatusCode;
                    response.Statusmsg = r.StatusDescription;
                    response.Content = Utils.StringFromStream(r.GetResponseStream());
                    r.Close();
                }
                catch (WebException e)
                {
                    sw.Reset();
                    if (e.Status == WebExceptionStatus.Timeout)
                    {
                        Global.Settings.Timeouts++;
                        request.Abort();
                        continue;
                    }

                    response.Ex = e;
                    HttpWebResponse r = (HttpWebResponse)e.Response;
                    if (r != null)
                    {
                        response.Statuscode = (int)r.StatusCode;
                        response.Statusmsg = r.StatusDescription;
                        response.Content = Utils.StringFromStream(r.GetResponseStream());
                        r.Close();
                    }
                    throw;
                }
                return response;
            }
            throw new Exception("HTTP request has timed out" + (resend + 1) + " times in a row
                . Check your internet connection.");
        }
```

Výpis 6: Metoda Net.GetResponse

6.5 Práce s Google API

Aby mohla aplikace komunikovat s Google Calendar a Google Tasks API, bylo potřeba implementovat zaprvé objekty reprezentující jednotlivé prostředky dané API, a zadruhé metody pro komunikaci s API.

Objekty i metody byly vytvořeny dle speficikací dostupných v oficiální dokumentaci Google API [2, 3]. Jako názvovou konvenci pro objekty jsem zvolil přidání písmena G před název prostředku, který reprezentuje. Všechny objekty obsahují properties definované v popisech korespondujících prostředků Google API uvedených v přílohách B a C, proto je také nebudu již znovu popisovat zde.
Pro Google Calendar byly vytvořeny následující objekty:

- GCalendarList
- GCalendarListEntry
- GEvents
- GEvent

a následující statické třídy obsahující metody pro komunikaci s API:

- GCalendarListRes
  - List
- GEventsRes
  - List
  - Insert
  - Delete

Pro Google Tasks byly vytvořeny následující objekty:

- GTaskLists
- GTaskList
- GTasks
- GTask

a následující statické třídy obsahující metody pro komunikaci s API:

- GTaskListsRes
  - List
- GTasksRes
  - List
  - Insert
  - Delete

Pro znázornění toho, jak vypadá třída pro komunikaci s API, si uvedeme třídu GEventsRes:

```
namespace Firesync.GoogleCalendar
{
    public static class GEventsRes
    {
        private const string BaseUri = "https :// www.googleapis.com/calendar/v3";

        public static GEvents List(string calendar_id)
        {
            RequestContainer rcon = Net.Get(BaseUri + "/calendars/" + Uri.EscapeDataString(
                calendar_id) + "/events", null, true, true);
            return Utils.JDeserialize<GEvents>(Net.GetResponse(rcon).Content);
        }
        public static NetResponse Insert(string calendar_id, GEvent gevent)
        {
            string content = Utils.JSerialize(gevent);
            RequestContainer rcon = Net.Post(BaseUri + "/calendars/" + Uri.EscapeDataString(
                calendar_id) + "/events", content, true, true, ContentType.Json);
            return Net.GetResponse(rcon);
        }
        public static NetResponse Delete(string calendar_id, string event_id)
        {
            RequestContainer rcon = Net.Delete(BaseUri + "/calendars/" + Uri.EscapeDataString(
                calendar_id) + "/events/" + event_id, null, true, true);
            return Net.GetResponse(rcon);
        }
    }
}
```

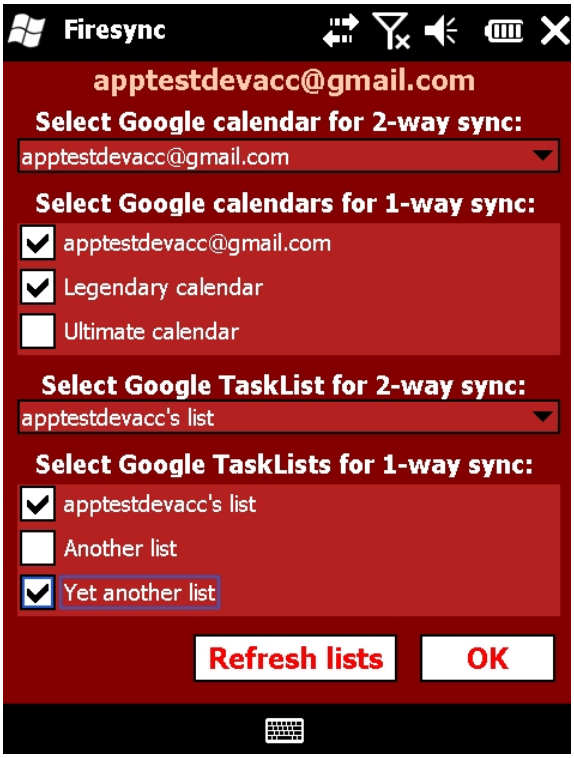
Výpis 7: Třída GEventsRes

Veškeré URI a požadované HTTP metody pro každou metodu jsou pevně dány dokumentací k API. Funkce objektů RequestContainer a NetResponse a metod třídy Net byly již důkladně popsány v předchozí sekci. Funkce metody Utils.JSerialize je přirozeně pravým opakem metody Utils.JDeserialize, tedy vytvoření JSON řetězce serializací objektu.

Význam jednotlivých metod pro práci s API je zřejmý:

- List vrací seznam všech událostí v kalendáři specifikovaném argumentem calendar_id.
- Insert vkládá do speficikovaného kalendáře novou událost, deserializovanou z objektu GEvent.
- Delete smaže speficikovanou událost v daném kalendáři.

Třída GCalendarListRes obsahuje pouze bezargumentovou metodu List, která vrátí seznam veškerých kalendářů uživatele.
Třídy GTasksRes a GTaskListsRes jsou analogickými ekvivalenty.



Obrázek 13: Menu výběru cílových prostředků

6.6 Výběr cílových prostředků synchronizace

Po úspěšné inicializaci autorizačního objektu Auth (ať už kompletní autentizací nového uživatele nebo prostým načtením tokenů ze souboru) lze již komunikovat s vlastními Google API. Toho Firesync také okamžitě využívá a stahuje do telefonu seznamy všech kalendářů a seznamů úkolů. Ukážeme si, jak vypadá proces stažení a zobrazení seznamu kalendářů uživatele.

Seznamu kalendářů odpovídá prostředek CalendarList. Nahlédnutím do referenční příručky GCal API zjistíme, že pro získání všech položek prostředku CalendarList (tedy všech kalendářů) se na prostředek zavolá metoda nazvaná *list*. Tu my máme již v aplikaci implementovánu jako GCalendarListRes.List(), stejně jako objekt reprezentující CalendarList, GCalendarList. Stačí tedy metodu zavolat.

Jak je možno vyčíst z přílohy, objekt GCalendarList je v podstatě pouze kolekcí objektů GCalendarListEntry, což jsou již samotné kalendáře. Z těchto pak můžeme vyčíst názvy jednotlivých kalendářů a zobrazit je všechny uživateli pro možnost výběru. Analogicky je tomu u úkolů, jedná se o objekty GTaskLists a GTaskList. Náhled menu je zachycen na obr. 13.

Po potvrzení výběru se do třídy Global uloží čtyři nové prvky:

- Objekt GCalendarListEntry, který uživatel vybral jako kalendář pro obousměrnou synchronizaci
- Kolekce objektů GCalendarListEntry, které uživatel vybral jako kalendáře pro jednosměrnou synchronizaci
- Objekt GTaskList, který uživatel vybral jako seznam úkolů pro obousměrnou synchronizaci
- Kolekce objektů GTaskList, které uživatel vybral jako seznamy úkolů pro jednosměrnou synchronizaci

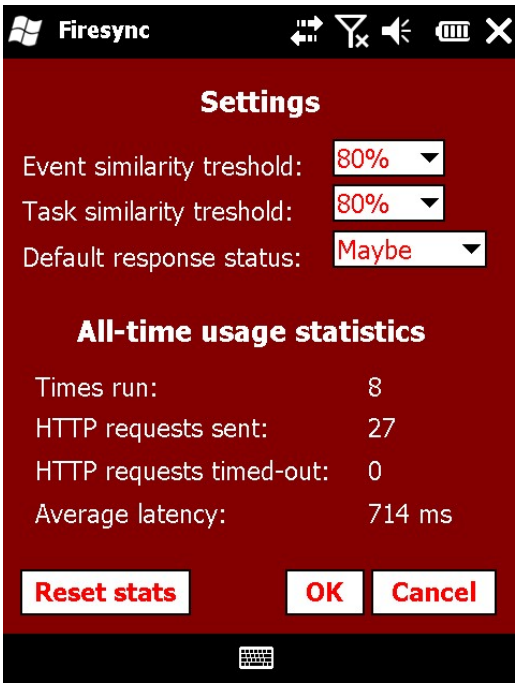
Tímto je proces výběru dokončen, volby jsou uloženy v globální třídě, a je již možno provést synchronizaci. Nejprve je ovšem třeba krátce popsat hlavní menu, ve kterém se po výběru prostředků ocitneme.

Obrázek 14: Hlavní menu

6.7 Hlavní menu a ostatní obrazovky

Hlavní menu (obr. 14) nabízí tyto možnosti:

- Settings and statistics - Vyvolá obrazovku nastavení a statistik (obr. 15). Význam jednotlivých položek byl již vysvětlen v rámci popisu třídy AppSettings v sekci 6.3.
- Change Google profile - Vyvolá dříve popsanou obrazovku pro autentizaci nového uživatele pomocí OAuth.
- About Firesync - Zobrazí informace o aplikaci a jejím autorovi (obr. 16).
- Exit - Uloží statistiky a ukončí aplikaci.
- Synchronize - Zobrazí synchronizační menu.



Obrázek 15: Nastavení a statistiky

Obrázek 16: Informace o aplikaci a autorovi

6.8 Synchronizace

Před popisem samotného synchronizačního procesu se nejdříve podíváme na dvě z jeho dílčích částí.

6.8.1 Konvertory a PocketOutlook API

Konvertory jsou nezbytnou součásti kterékoliv z variant synchronizace. Všechny se nacházejí ve třídě Converters a již od pohledu se dají rozdělit na dva druhy:

- **Primární konvertory**

  ty mají na starost konverzi celých objektů událostí a úkolů z Google formátu (tedy námi implementovaných objektů GEvent a GTask) na objekty, se kterými umí pracovat již samotný operační systém Windows Mobile. A samozřejmě i konverzi opačným směrem.

- **Sekundární konvertory**

  Jsou používány primárními konvertory k předvodu jednotlivých elementárních vlastností událostí a úkolů, případně jinými sekundárními konvertory k dílčím činnostem.

Před ilustrací toho, jak vypadá primární konvertor, nejprve pár slov o **PocketOutlook API**. Toto rozhraní, oficiálně referencované svým celým názvem PocketOutlook Object Model, neboli POOM, se používá k přístupu k PIM objektům uživatele. PIM znamená Personal Information Management, v praxi se jedná uživatelská data jako kontakty, úkoly, položky kalendáře a podobně. V případě POOM jsou podporovány právě tyto tři zmíněné typy PIM objektů a nás z toho samozřejmě zajímají dva, těmi jsou reprezentace události v podobně objektu Appointment a úkolu v podobě objektu Task.

Po prostudování dokumentace k PocketOutlook API se ukazuje, že její použití pro naše účely je v vcelku jednoduché. Krom objektů Appointment a Task (a jejich kolekcí AppointmentCollection a TaskCollection) už pracujeme pouze s objektem OutlookSession, který představuje rozhraní mezi naší aplikací a samotnými realnými událostmi a úkoly, které se nachazí v telefonu, jeho instance je vstupním bodem do celého modelu POOM.

U použití OutlookSession je třeba se vyvarovat jedné věci - a to je vytváření více instancí tohoto objektu. Oficiální dokumentace to sice nikde neuvádí, ale v opačném případě může docházet k neočekávaným výjimkám při následném přístupu k API objektům (tyto výjimky jsou navíc obecné a o ničem nevypovídající, jako třeba *Native method call failed*). My proto používáme v celé aplikaci jedinou instanci OutlookSession, kterou uchováváme ve třídě Global.

Získání kolekcí všech událostí a úkolů se pak provádí jednoduše:

```
OutlookSession osession = new OutlookSession();

AppointmentCollection appts = osession.Appointments.Items;
TaskCollection tasks = osession.Tasks.Items;
```

Výpis 8: Použití objektu OutlookSession

Nyní si již pro ilustraci primárního konvertoru ukážeme metodu GEventToAppt (GEvent to Appointment):

```
public static Appointment GEventToAppt(GEvent gevent)
{
    Appointment appt = new Appointment();
    appt.Subject = gevent.summary;
    appt.Body = gevent.description;
    appt.Location = gevent.location;

    if (gevent.AllDay)
    {
        appt.AllDayEvent = true;
        appt.Start = gevent.start.Date.Value;
        appt.End = gevent.end.Date.Value;
    }
    else
    {
        appt.AllDayEvent = false;
        appt.Start = gevent.start.dateTime.Value;
        appt.End = gevent.end.dateTime.Value;
    }

    foreach (GEvent.EventAttendee attnd in gevent.attendees)
        appt.Recipients.Add(new Recipient((attnd.displayName != null) ? attnd.displayName
            : "Unnamed", attnd.email));

    appt.ParseRRULE(gevent.recurrence);

    return appt;
}
```

Výpis 9: Metoda Converters.GEventToAppt

Pozn.: properties Google objektů jako je GEvent začínají úmyslně malým písmenem (přestože to odporuje konvenci), protože při serializaci do formátu JSON se do výsledného řetězce zaznamenává přesný název každé property. Pokud se na Google odešle serializovaná událost a ta obsahuje vlastnosti začínající velkým písmenem místo malého, Google vrací chybu.

Konverzní metoda samotná probíhá ve vcelku prostém duchu, properties objektu GEvent jsou postupně přirazeny jejich ekvivalentům objektu Appointment. Můžeme si ovšem všimnout volání Appointment.ParseRRULE, což je sekundární konvertor v podobě extension metody objektu Appointment.

Jednotlivé sekundární konvertory si popisovat nebudeme, vetšina z nich je zcela triviálních (např. převod řetězce názvu dne ve stylu MO, WE, FR.. na patřičnou hodnotu enumerace DayOfWeek {Monday, Tuesday..}). ParseRRULE je jedním z těch komplikovanějších a nyní si provedeme jeho popis.

RRULE je řetezec podléhající standardu iCalendar a představuje definici rekurenčního vzoru události. Může obsahovat následující parametry:

- FREQ - určuje frekvenci opakování (denně, týdně..)
- INTERVAL - určuje interval opakování v rámci frekvence (např. FREQ=WEEKLY a INTERVAL=2 znamená opakování každé dva týdny). Nepovinné.
- BYDAY - u týdenního opakování obsahuje seznam dnů v týdnu, kdy se má událost objevit (např. BYDAY=FR,SA). U měsíčního opakování určuje, který den měsíce dochází k opakování (např. BYDAY=3SA znamená třetí sobotu v měsíci). Nepovinné.
- UNTIL/COUNT - určuje konec opakování události, buď formou pevného data nebo celkového počtu opakování. Nepovinné. Najednou lze použít pouze jeden z těchto dvou parametrů.

Celý řetězec potom může vypadat třeba takto:
"RRULE:FREQ=DAILY;INTERVAL=2;COUNT=4". Metoda ParseRRULE přitom slouží k převodu řetězce RRULE na objekt RecurrencePattern, který je vlastností objektu Appointment a jak je z názvu zřejmé, definuje rekurenční vzor události.

```
public static Appointment ParseRRULE(this Appointment appt, List<string> recurence)
{
    AppointmentRecurrence apptrec = appt.RecurrencePattern;
    apptrec.RecurrenceType = RecurrenceType.NoRecurrence;
    if (recurence.Count == 0)
        return appt;

    Dictionary<string, string> rulenv = RRULEToDict(recurence);
```

Výpis 10: Metoda Converters.ParseRRULE - fragment 1

Nejpodstatnější částí tohoto fragmentu je volání dalšího sekundárního konvertoru, RRULEToDict. Jeho účelem je samotný řetězec RRULE převést na slovník obsahující jednotlivé parametry a hodnoty v pohodlně manipulovatelném formátu. Při tvobe slovníku je použito několikanásobného vnořeného volání metody Split(char), rozdělující řetězec podle zadaného znaku (v tomto případě tedy postupně podle dvojtečky, středníku a rovnítka).

```
if (rulenv["FREQ"] == "DAILY")
{
    apptrec.RecurrenceType = RecurrenceType.Daily;
}
if (rulenv["FREQ"] == "WEEKLY")
{
    apptrec.RecurrenceType = RecurrenceType.Weekly;
    if (rulenv.ContainsKey("BYDAY"))
    {
        string[] days = rulenv["BYDAY"].Split (',') ;
        if (days.Contains<string>("MO"))
            apptrec.DaysOfWeekMask |= DaysOfWeek.Monday;
...
```

Výpis 11: Metoda Converters.ParseRRULE - fragment 2

Dále již následuje postupné nastavení properties objektu RecurrenceType na základě hodnot ve vytvořeném slovníku. Zbytek metody vypada obdobně.

6.8.2 Komparátory

Účelem komparátorů je ohodnocení míry podobnosti dvou událostí či úkolů poměřením hodnot jejich elementárních vlastností. Průběh komparační metody je vcelku prostý - viz metoda poměřující dvě události typu GEvent:

```
public static int CompareGEvents(GEvent g1, GEvent g2)
{
    int score = 0;
    if (g1.summary == g2.summary || (g1.summary == null && g2.summary == "") || (g1.
        summary == "" && g2.summary == null))
        score++;
    if (g1.description == g2.description || (g1.description == null && g2.description == ""
        ) || (g1.description == "" && g2.description == null))
        score++;
    if (g1.location == g2.location || (g1.location == null && g2.location == "") || (g1.
        location == "" && g2.location == null))
        score++;

...

    return score;
```

Výpis 12: Metoda CalSync.CompareGevents

Skóre podobnosti se pohybuje v rozmezí od 0 do 10 u událostí a od 0 do 5 u úkolů, přičemž maximální hodnota znamená v obojím případě shodu. Výsledek u úkolů je přitom vážený - protože zdaleka nejpodstatnější (a často i jedinou) vlastností úkolu je jeho název, má větši vliv na výsledné skóre, než ostatní vlastnosti.

Obrázek 17: Synchronizační menu

6.8.3 Průběh synchronizace

Vlastní synchronizace začíná, jakmile uživatel vybere jednu ze čtyř synchronizačních variant (události jednosměrně a obousměrně, úkoly jednosměrně a obousměrně). Synchronizační menu na obr. 17 také nabízí možnost přejít na dříve popsanou obrazovku s výběrem cílových kalendářů a seznamů úkolů, kliknutím na tlačítko "Select target Calendars and TaskLists".

Pro demonstraci průběhu synchronizačního procesu použijeme metodu CalSync.Sync, která obstarává obousměrnou i jednosměrnou synchronizaci kalendáře (jednosměrná synchronizace s více kalendáři najednou je realizována cyklickým voláním této metody pro každý kalendář). Metoda TaskSync.Sync funguje v podstatě naprosto identicky.

```
public static void Sync(GetDecision showDialog, GCalendarListEntry gcal, SyncDirection
    direction)
{
    string calId = gcal.id;

    Utils.SetMessage("Downloading events", true);
    GEvents allGevents = GEventsRes.List(calId);

    GEvents allWevents = new GEvents();
    AppointmentCollection appts = Global.Appointments();
    foreach (Appointment appt in appts)
        allWevents.items.Add(Converters.ApptToGEvent(appt));
```

Výpis 13: Metoda CalSync.Sync - fragment 1

Prvním argumentem metody je delegát typu GetDecision. Ten je použit ve třetí fázi synchronizace (detekce možných aktualizací) kde je potřeba zpětná vazba ze strany uživatele. Knihovna FiresyncLib je ovšem oddělena od uživatelského rozhraní a nemůže dotazovací dialog vyvolat přímo, využíva se proto delegace. Tento proces je (stejně jako delegát samotný a objekt, který je v jeho volání předáván) popsán níže ve zmíněné části synchronizace.

Zbývající argumenty již nepotřebují širší popis, jsou jimi objekt typu GCalendarListEntry určující, se kterým kalendářem má synchronizace proběhnout a enumerace SyncDirection (s možnými hodnotami Bidirectional a GoogleToDevice) rozhodující, zda jde o obousměrnou či jednosměrnou synchronizaci.

1. fáze - získání dat
Metoda Utils.SetMessage slouží k aktualizaci textu zobrazeného ovládacího prvku informujícího uživatele o průběhu akce - v našem případě se tedy jedná o dříve zmíněný ProgressBox. Druhý argument metody určuje, zda má být zpráva statická(false) či animovaná. Metoda Utils.SetMessage samozřejmě taktéž volá pouze delegát inicializovaný uživatelským rozhraním.

Dalším krokem je již získání všech událostí z přislušného Google kalendáře. API dokumentace prozradí, že k tomu se používá metoda *list* zavolaná na prostředek Events. My tedy použijeme naši korsepondující metodu, GEventsRes.List() a získaná data uložíme do objektu allGevents typu GEvents.

Následně vytvoříme další instanci objektu GEvents pojmenovanou allWevents (Windows events) a do jeho kolekce přidáme všechny lokální události převedené do formátu GEvent. Důvod k tomuto je prostý - námi vytvořený objektový typ GEvent (a samozřejmě analogicky i GTask) totiž obsahuje krom vlastností vyplývajících z Google Calendar prostředku Event i dvě vlastnosti nutné pro chod synchronizace (tyto se samozřejmě při převodu do JSON řetězce neserializují a na Google servery nejsou posílány).

Těmito vlastnostmi jsou:

- SyncState - enumerace nabývající hodnot NotSet, Ignore, Delete, Store. Během druhé a třetí synchronizační fáze je GEvent objektům hodnota této vlastnosti adekvátně nastavena a na jejím základě je v poslední, čtvrté fázi následně provedena příslušná akce. Výchozí hodnota je samozřejmě NotSet.
- ComparisonResults - objekt, jehož účelem je uchovávat informace o tom, jaké události jsou této GEvent instanci podobné. Ve druhé fázi jsou tyto informace vloženy a ve třetí následně použity k dotázáni uživatele, zda je podobnost náhodná, či zda se jedná o aktualizaci. Využito pouze při obousměrné synchronizaci (jednosměrná třetí fázi přeskakuje).

2. fáze - komparace
Další fragment zdrojového kódu ukazuje podobu celého komparačního algoritmu.

```
Utils.SetMessage("Comparing events", true);
foreach (GEvent wevent in allWevents.items)
{
    foreach (GEvent gevent in allGevents.items)
    {
        if (gevent.SyncState == SyncState.Ignore)
            continue;
        int result = CompareGEvents(gevent, wevent);
        if (result == 10)
        {
            gevent.SyncState = SyncState.Ignore;
            wevent.SyncState = SyncState.Ignore;
            break;
        }
        if (result >= Global.Settings.EventSimilarityTreshold)
            gevent.ComparisonResults.Add(wevent, result);
    }
}
```

Výpis 14: Metoda CalSync.Sync - fragment 2

Vnější cyklus *foreach* iteruje veškeré lokální události, vnitřní cyklus události vzdálené (stažené z Google). První příkaz vnitřního bloku zajišťuje přechod k další vnitřní iteraci v případě, že daná vzdálená událost byla již shledána synchronizovanou. V opačném případě je zavolána komparační metoda CompareGEvents a její výsledek uložen do celočíslené proměnné.

Pokud je nalezena shoda (tedy výsledek komparace se rovná 10, v případě úkolů 5), lokální i vzdálené události je nastavena vlatnost SyncState na hodnotu Ignore, což znamená, že události jsou synchronizované a není potřeba žádné akce.

Pokud není nalezena shoda zjišťuje se, zda jsou si události alespoň dostatečně podobné. Potřebná hranice pro tento stav je nastavena uživatelem. Pokud si podobné jsou, do objektu ComparisonResults embedovaného ve vzdálené události je vložena reference na místní (podobnou) událost, společně s výsledkem komparace.

3. fáze - zpracování podobných objektů

Pokud probíhá jednosměrná synchronizace, tento krok je přeskočen a spouští se poslední fáze:

```
if (direction == SyncDirection.GoogleToDevice)
{
    Utils.SetMessage("Applying changes", true);
    ProcessGevents(allGevents.items, calId, direction);
    return;
}
```

Výpis 15: Metoda CalSync.Sync - fragment 3

Metoda ProcessGevents bude popsána v rozboru poslední fáze.
Pokud se jedná o obousměrnou synchronizaci, provede se procedura zpracování podobných událostí za účelem určení aktualizací, na základě vstupů uživatele.

První částí této fáze je příprava dat:

```
List<GEvent> decisionRequired = new List<GEvent>();
foreach (GEvent gevent in allGevents.items)
{
    if (gevent.SyncState != SyncState.Ignore)
    {
        gevent.ComparisonResults.PurgeProcessed();
        if (gevent.ComparisonResults.HasData())
            decisionRequired.Add(gevent);
        else
            gevent.SyncState = SyncState.Store;
    }
}
```

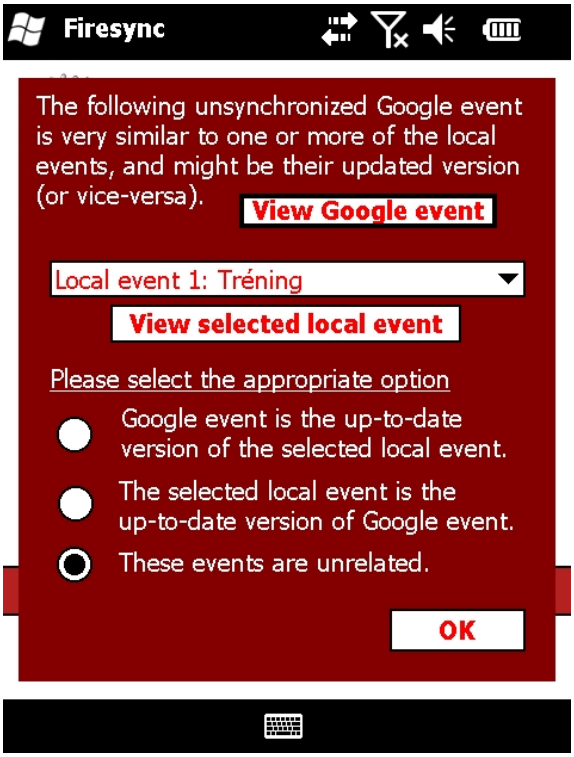
Výpis 16: Metoda CalSync.Sync - fragment 4

Některé vzdálené události mohou mít v seznamu podobných místních událostí i ty, které jsou shodné s jinou vzdálenou událostí (tzn. místní událost má SyncState nastaveno na Ignore). K tomu může dojít tak, že při komparačním procesu je neprve nalezena podobnost, místní událost je přidána do seznamu podobností ve vzdálené události, ovšem místní událost je dále poměřována s ostatními vzdálenými událostmi a je nalezena shoda s jinou událostí.

Pro odstranění těchto podobných událostí (které nemohou být aktualizacemi, protože mají již shodu v jiné události) ze seznamu ComparisonResults se používá metoda PurgeProcessed(), která ze seznamu podobností vymaže všechny události, jejichž SyncState se nerovná NotSet.

Pokud po této operaci nemá vzdálená událost v seznamu podobností již žádné položky, znamená to, že musí být synchronizována a její SyncState je nastavena na Store. V opačném případě je událost přidána do kolekce decisionRequired, což je seznam vzdálených událostí, které mohou být potenciálním předmětem aktualizace.

V druhé části této fáze je iterována kolekce událostí decisionRequired a uživateli jsou zobrazeny potenciální možnosti aktualizace. Ke zobrazení dialogu pro interakci se



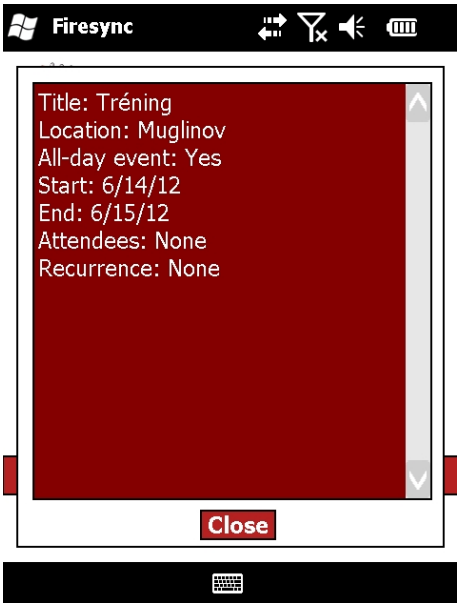
Obrázek 18: Dialog pro posouzení potenciální aktualizace

používá delegát GetDecision, jež vrací void a jehož volání obsahuje argument typu UpdateDecisionQuery. Tento objekt je použit jak pro předání informací do vsrty uživatelského rozhraní (potřebných pro vytvoření dotazovacího dialogu), tak k získání zpětné vazby z uživatelského rozhraní.
Objekt UpdateDecisionQuery má následující vlastnosti:

- GEvent Gevent - vzdálená událost
- List<GEvent>Locals - kolekce místních událostí podobných vzdálené události
- UpdateDecision Decision - enumerace obsahující GoogleIsUpdate, LocalIsUpdate, Unrelated
- GEvent LocalUpdate - v případě, že uživatel označí jednu z místních událostí jako aktualizaci (případně naopak, jako zastaralou událost), je do této vlastnosti uložena reference na ni

Dialog pro posouzení podobnosti je k vidění na obrázku 18, okno s detailem události (které se zobrazí po kliknutí na tlačítka ”View Google event”nebo ”View selected local event”) pak na obrázku 19.

Poté co je dialog zavřen, dochází na základě rozhodnutí uživatele (reprezentovaného hodnotou objektu enumerace UpdateDecision) k jedné z následujících akcí:



Obrázek 19: Detail události v dialogu

- **Uživatel určil vzdálenou událost jako aktualizaci jedné z místních událostí:** vzdálené události je SyncState nastaven na Store, místní na Delete
- **Uživatel určil jednu z místních událostí jako aktualizaci vzdálené události:** vzdálené události je SyncState nastaven na Delete, místní na Store
- **Uživatel určil, že se jedná o separátní události:** vzdálené události je SyncState nastaven na Store, všechny místní události zůstávají ve stavu NotSet (a mohou být znovu zobrazeny v dalším dialogu pro jinou vzdálenou událost)

4. fáze - aplikování změn
Nyní v sobě již všechny objekty události mají nastavenu patřičnou hodnotu SyncState a zbývá pouze provést odpovídající akci pro každou událost. K tomu slouží metody ProcessGevents a ProcessWevents. Jejich argumenty jsou:

- kolekce událostí pro zpracování
  -metodě ProcessGevents se samozřejmě předá kolekce allGevents a metodě ProcessWevents kolekce allWevents.
- cílový Google kalendář
- SyncDirection

Pokud probíhá jednosměrná synchronizace, volá se pouze metoda ProcessGevents. Jaké akce jsou pro jednotlivé události v těchto metodách provedeny na základě jejich SyncState je uvedeno v tabulce 1.

SyncState	ProcessGevents	ProcessWevents
Store	Uloží událost do přistroje	Odešle událost přes Google API
Delete	Smaže událost přes Google API	Smaže událost z přístroje
Ignore	Žádná akce	Žádná akce
NotSet	Totéž jako při Store	Totéž jako při Store

Tabulka 1: Porovnání synchronizačních akcí

Doplňující informace:

- Smazání přes Google API je realizováno metodou GEventsRes.Delete
- Odeslání události přes Google API neboli přidání do vzdáleného kalendáře je realizováno metodou GEventsRes.Insert
- Pokud jde o obousměrnou synchronizaci, události v metodě ProcessGevents nemohou nikdy mít SyncState o hodnotě NotSet. Vyplývá to z algoritmu synchronizace.

6.9 Multithreading

Na závěr komentář k použití více vláken v prostředí Windows Forms, čímž byl podstatně ovlivněn vývoj aplikace.

Provádění časově náročných operací v UI vlákně aplikace (ve výchozím stavu má aplikace pouze toto vlákno) má za následek neresponsivnost až úplné zamrznutí rozhraní, trvající dokud není akce dokončena. To znemožňuje aktualizovat rozhraní v průběhu složitějších procedur, čili animace našeho objektu ProgressBox by například během synchronizace kalendáře nebyla možná.

Jako nejjednodušší řešení se nabízí zkrátka spouštět takovéto procedury v nových, separátních vláknech. Vskutku, tímto krokem je jev zamrzání uživatelského rozhraní eliminován. Zahrnuje ovšem i jedno velmi podstatné úskalí - platí totiž, že k prvkům uživatelského rozhraní je dovoleno přistupovat pouze z vlákna, kde byly vytvořeny.

Důvodem pro toto je koncept nazvaný *thread safety* (bezpečnost vláken) [10]. Procedura nebo volání je považováno za non thread-safe, pokud manipuluje s cizím vláknem takovým způsobem, který může jeho data přivést do nekonzistentního stavu. Právě to hrozí prvkům uživatelského rozhraní, pokud je k nim přistupováno ze dvou nebo více vláken (včetně jejich vlastního vlákna) najednou, a proto to framework nedovoluje a při pokusu o nebezpečný přístup vyvolá výjimku.

Naštěstí existuje řešení, a to v podobě metody Control.Invoke. Tato metoda je vždy thread-safe (lze ji tedy volat i z jiných vláken) a jejím účelem je provést definovanou akci na vlákně, kterému objekt Control (neboli prvek UI) náleží. Požadovaná akce je metodě předána argumentem ve formě delegátu.

Pro účely použití multithreadingu a thread-safe přístupu v mé aplikaci jsem sestavil několik metod, které tyto techniky ulehčují. První z nich je Utils.MultithreadExec:

```
public static Thread MultithreadExec(string threadName, Action action)
{
    Thread t = new Thread(delegate { action.Invoke(); });
    if (threadName != null)
        t.Name = threadName;

    t.Start();
    return t;
}
```

Výpis 17: Metoda Utils.MultithreadExec

Metodě je předána požadovaná procedura ve formě generického delegáta Action a dochází k jejímu provedení v novém vlákně. Volání metody vypadá například takto:

```
Utils.MultithreadExec("FetchTokens", delegate
{
    Auth.FetchTokens(code);
    ProgressBox.StaticMessage("Success!");
    UI_Utils.ClearUserData();
    Thread.Sleep(2000);

    ProgressBox.HideNow();
    Window.NextScreen();
}
);
```

Výpis 18: Použití metody Utils.MultithreadExec

Uvedený kód je použit při OAuth autentizaci poté, co uživatel vloží a potvrdí autorizační kód zpět do aplikace. Všimněme si, že metody ProgressBox.StaticMessage a ProgressBox.HideNow nejsou použity s pomocí zmíněné thread-safe metody Invoke. Jedná se totiž a statické metody, a až ty samotné následně manipulují se skutečnou instancí ProgressBoxu - volání Invoke obstarávají samy, pokud je potřeba. Viz náhled metody ProgressBox.HideNow:

```
public static void HideNow()
{
    if (! UI_Utils.InvokeRequired())
        Window.Progress.HideNowProcess();
    else
        UI_Utils.UIThreadExec(delegate { Window.Progress.HideNowProcess(); });
}
```

Výpis 19: Metoda ProgressBox.HideNow

Window.Progress je globální reference na ProgressBox objekt. HideNowProcess je už samotná metoda, která objekt skryje.

Jak je vidět, pokud Invoke není potřeba použít (což lze determinovat pomocí InvokeRequired), metoda HideNowProcess je zavolána přímo. V opačném případě je delegována jako argument metody UI_Utils.UIThreadExec, která spouští danou akci v UI vlákně. Tato metoda pouze zavolá UIThreadHandle.Invoke(Action), kde UIThreadHandle je globální reference na hlavní okno aplikace, které vždy běží v hlavním UI vlákně.

6.10 Shrnutí úprav nutných pro přenesení na platformu Windows Phone 7

Jak již bylo zmíněno v analýze mobilních platforem Windows, při přenesení na WP7 je potřeba vytvořit od základu nové UI. Naštěstí se nám podařilo UI od aplikační knihovny oddělit v podstatě dokonale, takže tento proces nebude ničím komplikován.

Krom nového uživatelského rozhraní bude potřeba nahradit ještě několik dalších objektů, které se na obou platformách implementují odlišně. Jejich množství se podařilo udržet na, myslím si, velmi přijatelné úrovni:

- **Třída TimeZone**
  Musí být nahrazena třídou TimeZoneInfo.
- **Třída ServicePointManager**
  Nemá na platformě WP7 ekvivalent. Její použití v aplikaci naštěstí není vůbec zásadní.
- **Třída StopWatch**
  Na platformě WP7 neexistuje, může být nahrazena například manuálním měřením času pomocí DateTime.Now.Ticks a třídy TimeSpan. Tuto implementaci jsem se kvůli kompatibilitě pokoušel použít již v mnou vytvořené verzi aplikace, ovšem ukázalo se, že objekty DateTime ve WM6.5 (přinejmenším v emulátoru) nejsou schopny měřit milisekundy a tedy ani Ticks.
- **Metoda Control.Invoke**
  Musí být nahrazena metodou Dispatcher.BeginInvoke.
- **Třída HttpWebRequest**
  Podporuje pouze asynchronní verze metod GetRequestStream a GetResponse (BeginGetRequestStream/EndGetRequestStream a BeginGetResponse/EndGetResponse), kterými musí být nahrazeny.

7 Závěr

Cílem práce bylo analyzovat existující softwarová řešení pro synchronizaci mobilních platforem Microsoft Windows Mobile 6.5 a Windows Phone 7 se službou Google Calendar a následně navrhnout a implementovat vlastní řešení pro systém Windows Mobile 6.5 takovým způsobem, aby byla aplikace co nejjednodušejí přenesitelná na platformu Windows Phone.

Nejprve jsem provedl rozbor relevantních služeb a technologií společnosti Google a následně mobilních Windows platforem. Během něj se ukázalo, že pro systém Windows Phone 7 bohužel není momentálně možné aplikaci s požadovanými parametry vytvořit. Tento stav ovšem nemusí být trvalý, takže na způsobu mého postupu návrhu a implementace se tím nic nezměnilo. Z analýzy rozdílů mezi platformami Windows Mobile 6.5 a Windows Phone 7 dále vyplynulo, že pro co největši usnadnění případného přenesení aplikace bude nutné co možná nejdůsledněji oddělit uživatelské rozhraní od aplikačně-datové části řešení a v implementaci preferovat použití těch objektů, které jsou dostupné na obou platformách.

Testování tří volně dostupných softwarových synchronizačních řešení na platformě Windows Mobile 6.5 ukázalo, že ani jedno nefunguje spolehlivě. Na vině může být fakt, že všechna byla vytvořena pro starší verze platformy nebo také použití zastaralé verze Google API. Nicméně, z analýzy jsem vyvodil důsledky ve formě požadavků na mou aplikaci eliminujících zjištěné nedostatky analyzovaných řešení.

Následoval návrh vlastního řešení na základě všech získaných poznatků a poté jeho implementace. Obojí proběhlo úspěšně a výsledná aplikace, pojmenovaná Firesync, splňuje všechny prvky zadání práce. Dosavadní testování ukázalo, že synchronizace kalendářů i úkolů probíhá korektně a spolehlivě.

Finální implementace obsahuje pouze pět objektů, které by při přenesení na platformu Windows Phone bylo potřeba nahradit či odstranit, což by měl být více než uspokojivý výsledek.

Do budoucna plánuji aplikaci rozšířit o možnost importování dat kalendáře ze souboru ve formátu iCalendar a přidat funkci mazání událostí a úkolů (tak, aby se nedostaly zpět při synchronizaci).

8 Reference

[1] Google Inc., *Using OAuth 2.0 to Access Google APIs*, [online], 2011.
Dostupné z WWW: https://developers.google.com/accounts/docs/OAuth2

[2] Google Inc., *Google Calendar API v3*, [online], 2011.
Dostupné z WWW: https://developers.google.com/google-apps/calendar/

[3] Google Inc., *Google Tasks API v1*, [online], 2011.
Dostupné z WWW: https://developers.google.com/google-apps/tasks/

[4] Google Inc, *Google Data APIs*, [online], 2005.
Dostupné z WWW: https://developers.google.com/gdata/

[5] Richardson L., Ruby S., *RESTful Web Services*, O'Reilly Media, 2007,
ISBN 978-0-596-52926-0

[6] Microsoft Corporation, *.NET Compact Framework 3.5*, [online], 2008.
Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/f44bbwa1(v=vs.90).aspx

[7] Microsoft Corporation, *Windows Mobile 6.5*, [online], 2010.
Dostupné z WWW: http://msdn.microsoft.com/en-us/library/bb158486.aspx

[8] Microsoft Corporation, *Windows Phone Development*, [online], 2012.
Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/ff4025359(v=vs.92).aspx

[9] Microsoft Corporation, *The Silverlight and XNA Frameworks for Windows Phone*,
[online], 2012. Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/ff402528(v=vs.92.aspx

[10] Microsoft Corporation, *Thread-Safe Calls to Windows Forms Controls*,
[online], 2008. Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/ms171728(v=vs.90).aspx

A Obsah CD

- Složka **bin** - obsahuje zkompilované soubory aplikace vč. instalátoru
- Složka **doc** - obsahuje tento dokument
- Složka **res** - obsahuje veškeré grafické soubory
- Složka **src** - obsahuje zdrojový kód aplikace

B Google Calendar API - detail prostředků

Google Calendar API zahrnuje celkem 7 typů prostředků a mnoho metod pro každý typ (u prostředku Event dokonce 10 metod), uvádím zde pouze popis těch relevantních, tedy těch, které jsem využil při synchronizaci. Stejně tak popisuji pouze ty vlastnosti jednotlivých prostředků, které buď mají svůj ekvivalent v kalendáři na synchronizovaném zařízení nebo jsou jiným způsobem využity při synchronizačním procesu. U metod je kromě odpovídajícího typu HTTP metody uveden i alias používaný v rámci API, a zároveň i v implementaci aplikace.

B.1 Prostředek CalendarList

Reprezentuje seznam všech kalendářů v Google účtu.
Vlastnosti:

- kind - identifikuje typ prostředku
- items - kolekce prostředků CalendarListEntry

Metody:

- GET (aka list) - vrátí seznam kalendářů v Google účtu.

B.2 Prostředek CalendarListEntry

Reprezentuje jeden kalendář v Google účtu.
Vlastnosti:

- kind - identifikuje typ prostředku
- id - unikátní identifikátor kalendáře
- summary - název kalendáře
- description - popis kalendáře
- location - umístění kalendáře v reálném světě
- timeZone - časová zóna kalendáře

Metody:

- (žádné)

B.3 Prostředek Events

Reprezentuje všechny události v rámci jednoho kalendáře.
Vlastnosti:

- kind - identifikuje typ prostředku
- items - kolekce prostředků Event

Metody:

- GET (aka list) - vrátí seznam všech událostí v kalendáři.
- POST (aka insert) - vloží novou událost do kolekce

B.4 Prostředek Event

Reprezentuje jednu událost.
Vlastnosti:

- kind - identifikuje typ prostředku
- id - unikátní identifikátor události
- summary - název události
- description - popis události
- location - místo události
- start - počáteční datum (příp. i čas) události
- end - konečné datum (příp. i čas) události
- recurrence - řetezec popisující vzor rekurence události
- attendees - kolekce účastníků události
  - displayName - jméno účastníka
  - email - email účastníka
- status - stav události

Metody:

- DELETE (aka delete) - smaže událost

C Google Tasks API - detail prostředků

GTasks API zahrnuje celkem 4 typy prostředků, při synchronizaci využívám všechny. U metod a vlastností prostředků se držím stejné zásady jako u kalendáře, tedy zmiňuji je pouze pokud jsou použity pří synchronizaci. U metod je kromě typu HTTP metody uveden i alias používaný v rámci API, a zároveň i v implementaci aplikace.

C.1 Prostředek TaskLists

Reprezentuje seznam všech seznamů úkolů v Google účtu.
Vlastnosti:

- kind - identifikuje typ prostředku
- items - kolekce prostředků TaskList

Metody:

- GET (aka list) - vrátí seznam všech seznamů úkolů v Google účtu.

C.2 Prostředek TaskList

Reprezentuje jeden seznam úkolů v Google účtu.
Vlastnosti:

- kind - identifikuje typ prostředku
- id - unikátní identifikátor seznamu úkolů
- title - název seznamu úkolů

Metody:

- (žádné)

C.3 Prostředek Tasks

Reprezentuje všechny úkoly v rámci jednoho seznamu úkolů.
Vlastnosti:

- kind - identifikuje typ prostředku
- items - kolekce prostředků Task

Metody:

- GET (aka list) - vrátí seznam všech úkolů v seznamu
- POST (aka insert) - vloží nový úkol do kolekce

C.4 Prostředek Task

Reprezentuje jeden úkol.
Vlastnosti:

- kind - identifikuje typ prostředku
- id - unikátní identifikátor úkolu
- title - název úkolu
- notes - poznámky k úkolu
- status - stav úkolu
- due - termín úkolu (datum)
- completed - datum, kdy byl úkol splněn

Metody:

- DELETE (aka delete) - smaže úkol